VNU Journal of Science: Comp. Science & Com. Eng., Vol. 42, No.1 (2026) 79–114

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

A Semantic Framework and Tool Support for Unified Executable Domain Models in UDML: A Case Study on the RBAC Concern

Van-Vinh Le[1,2], Nhat-Hoang Nguyen[1], Duc-Quyen Nguyen[1], Duc-Hanh Dang[1*]

[1] *VNU University of Engineering and Technology, Hanoi, Vietnam*
[2] *Vinh University of Technology Education*

Received 04th February 2026
Revised 18th March 2026; Accepted 26th March 2026

Abstract: In Domain-Driven Design (DDD), domain models play a central role in capturing domain knowledge and aligning domain experts with software developers. Although UML/OCL and domain-specific languages (DSLs)—particularly annotation-based DSLs—enhance expressiveness for structural and behavioral modeling, they generally lack formally defined executable semantics. Consequently, domain models that integrate multiple concerns from different perspectives are prone to semantic ambiguities and provide limited support for systematic correctness analysis. In contrast, formal methods offer mathematically precise semantics and proof-based verification, but remain largely disconnected from concern-oriented and annotation-driven domain modeling practices. This paper introduces a unified semantic framework, together with tool support, for executable domain models. Specifically, we define a language called UDML (Unified Domain Model Language) to represent domain models in which concerns specified by corresponding DSLs are composed and share a common system state. We focus on three key concerns—structural constraints, domain behavior, and access policies—and define three corresponding DSLs: DCSL, AGL, and RBACDom. We then provide a semantic framework for the unified domain model and a transformation to Event-B. This framework enables the characterization and verification of domain model executions, while ensuring model consistency. We validate our approach through a realistic case study based on Open Journal Systems (OJS), demonstrating the feasibility and effectiveness of the proposed unified executable domain modeling framework.

Keywords: Domain Driven Design, Domain Specific Language, Event-B, UDML, RBAC.

*Corresponding author.
E-mail address: hanhdd@vnu.edu.vn
https://doi.org/10.25073/2588-1086/vnucsce.6743

1. Introduction

Domain-Driven Design (DDD) [1] advocates placing the domain model at the center of software development, with the expectation that such models accurately capture domain knowledge and guide implementation. In recent years, this vision has been strengthened by model-driven engineering (MDE) [2] approaches that aim to make domain models executable, thereby reducing the semantic gap between design-time abstractions and run-time systems. However, real-world domain models are rarely concerned with a single aspect. Instead, they typically encompass multiple concerns, including structural aspects, behavioral logic, and cross-cutting concerns such as security [3, 4].

A central challenge in DDD-based development is the definition of domain models that are both executable and semantically consistent when multiple concerns are involved. Although structural and behavioral aspects are often modeled separately, their execution semantics are intrinsically interdependent, and this interdependence becomes more critical in the presence of additional concerns such as access control, which constrain when and how domain behaviors may be executed. Without a unified semantic foundation, the integration of multiple concerns easily leads to inconsistencies, ambiguities, or unintended behaviors at execution time. To address executability, several approaches have proposed combining Domain-Specific Languages (DSLs) [5, 6], annotation-based techniques, and automated code generation [7]. However, existing solutions largely focus on individual concerns or isolated aspects of executability and provide limited support for ensuring semantic consistency across concerns. As a result, supporting executable, consistent, and multi-concern domain models remains a fundamental open problem in DDD-oriented model-driven development.

Our previous work introduced the Unified Domain Model Language (UDML) as a means to compose multiple domain concerns within a single executable model [8]. UDML demonstrates that structural and behavioral concerns can be modularly specified using dedicated DSLs and integrated through annotation-based composition, supported by a toolchain that enables model execution. However, existing UDML-based approaches focus primarily on engineering mechanisms for concern composition and code generation, but do not define a formal semantic foundation that characterizes how multiple concerns interact during execution. In particular, the semantics of domain behavior under additional constraints—such as security or authorization policies—remains implicit and tool-dependent.

To address this gap, this paper proposes a semantic framework that equips UDML with formal semantics, enabling the systematic integration of annotation-based domain concerns. Event-B [9–12] is adopted as the semantic foundation of UDML, providing a mathematically precise basis for defining and reasoning about domain execution. Behavioral aspects specified in AGL define the admissible domain state transitions and form the semantic backbone of the unified domain model. On this basis, we define RBACDom, a security concern expressed as an annotation-based DSL for RBAC, whose authorization rules are interpreted as guards constraining these behaviorally defined transitions. Structural, behavioral, and security concerns—captured respectively by DCSL, AGL, and RBACDom—are thus synchronized over a concern system state, yielding a unified transition system in which executability, cross-concern consistency, and correctness properties can be formally analyzed.

This paper extends our previous unified domain model approach (UDML) [8] by providing a formal semantic foundation and verification support for executable multi-concern domain models. The main contributions of this work are as follows:

- A semantic framework that provides means to explain the operational semantics of the unified domain model.
- A DSL, named RBACDom, to represent role-based access control as a concern incorporated into the unified domain model.
- A transformation from UDML models into Event-B, enabling proof-based verification using the Rodin platform.
- An empirical evaluation through a case study based on the Open Journal Systems (OJS) submission management process, including evaluation metrics for the generated proof obligations.

The paper is organized as follows. Section 2 presents the background and a motivating example; Section 3 provides an overview of our approach; Sections 4 and 5 present a semantic framework for unified domain models and a transformation from UDML to Event-B; Section 6 reports on tool support and case studies; Section 7 reviews related work; and Section 8 concludes the paper and outlines directions for future work.

2. Background and Motivation

This section motivates our work using a representative case study and introduces unified domain modeling. We focus on RBAC as a representative concern to demonstrate the necessity of concern incorporation within UDML and to motivate the need for formal semantics and verification of unified domain model.

2.1. Motivating Example

Figure 1 illustrates a simplified domain model of OJS, focusing on the closed-loop academic publishing workflow. The process begins with manuscript submission by an author, proceeds through multiple review and editorial decision

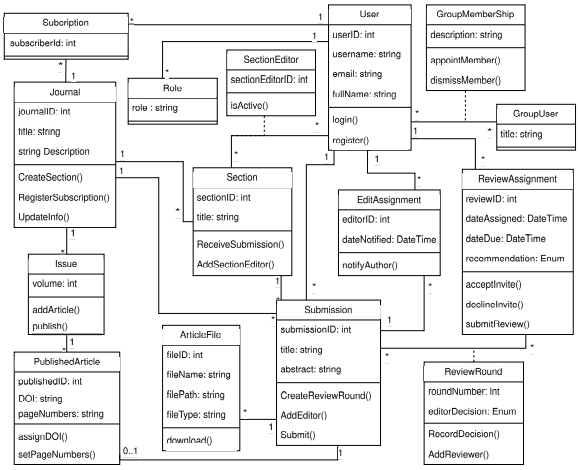

Figure 1. Domain model of Open Journal Systems.

stages, and concludes with publication and distribution to authorized subscribers.

The Open Journal Systems (OJS) [13] domain model encompasses multiple modeling aspects that are commonly addressed in software engineering practice [2], most notably structural elements and domain behavior, together with integrity constraints that capture essential domain rules. In our approach, structural constraints are specified using DCSL [14], which relies on essential OCL constraints to formalize domain structure within an executable model.

Example. An annotation-based DSL (aDSL) of DCSL is used to specify the structural constraints of the `User` class in OJS. The annotation `@DAttr(id = true, auto = true)` declares `id` as the identifier attribute of the user entity, with values automatically generated by the system. The annotation `@DAttr(optional = false, length = 100)` specifies that `email` is a mandatory attribute and enforces a maximum length constraint on its values.

```
public class User {
  @DAttr(id = true, auto = true)
  private int id;
  @DAttr(optional=false, length=100)
  private String email;
  ...
}
```

Beyond structural constraints, a key modeling challenge in the OJS domain arises from the dynamic and context-dependent nature of user privileges. In OJS, access rights are not statically bound to user identities but depend on both the target submission and the current stage of the editorial workflow. A single user may act as an author for one manuscript while simultaneously serving as a reviewer or a subscriber in other contexts. Consequently, traditional static modeling approaches based on role-specific subclasses (e.g., `Author` and `Reviewer`) are inadequate. Instead, the domain model must represent all actors using a unified `User` class, while deferring role interpretation and authorization decisions to runtime contexts.

Role-Based Access Control (RBAC) [15, 16] is a widely adopted access-control model in which permissions are associated with roles rather than individual users. According to the NIST RBAC standard [16], its core concepts include `users` as active system entities, `roles` representing job functions, `permissions` defined as action–resource authorizations, `sessions` capturing runtime role activation, and separation-of-duty (SoD) constraints—both static (SSD) and dynamic (DSD)—that prevent conflicts of interest.

While RBAC is conceptually well established, its integration into domain models remains challenging in complex, behavior-intensive systems such as OJS. In such domains, RBAC policies are tightly coupled with domain behavior and workflow states rather than being independent infrastructural policies. Authorization constraints therefore directly determine which domain operations are admissible at specific execution points. As a result, RBAC must be treated as a first-class behavioral concern, since access-control rules constrain state transitions and shape domain evolution. In OJS, authorization decisions depend not only on assigned roles but also on the current workflow stage, the target submission, and associated domain constraints. This tight coupling between behavior and authorization introduces significant semantic complexity in determining when and by whom domain operations may be executed.

Such characteristics make RBAC a particularly challenging concern to model. Approaches that treat authorization as an external or static mechanism are insufficient, as they fail to capture its tight coupling with behavioral states and domain execution constraints. This motivates the integration of RBAC into the OJS domain model through explicit specifications of access permissions, separation-of-duty constraints, and compliance requirements for each role, as summarized in Table 1.

2.2. Incorporating Concerns into Domain Models

To support rich domain modeling, various domain-specific languages (DSLs) have been proposed to explicitly capture structural constraints, business rules, and domain behavior [5, 6]. In this work, we consider three representative DSLs—DCSL [14], AGL [17], and UDML [8]—which together illustrate how typical structural and behavioral concerns can be composed within a unified domain model.

We have introduced UDML in [8] as a unifying metamodel layer that extends UML class diagrams with explicit support for concern composition. Its core metaconcepts—`DomainModel`, `Annotable`, `Concern`, and `Annotation`—enable concern-specific DSLs to be integrated through annotations without polluting or restructuring the core domain model. This design allows multiple heterogeneous concerns to coexist within a single domain model while preserving modularity and separation of concerns. However, UDML primarily addresses concern composition at the modeling level and does not ensure the feasibility of the resulting unified domain models at execution time. Moreover, it lacks a formal

Table 1. RBAC Specification for the OJS system

User	Least privilege principle	Separation of duties (SoD)	Compliance
U1 – Author	Submit manuscripts, upload revisions, manage metadata, and respond to reviews for own submissions only. No access to other submissions or reviewer identities.	Author must not activate Reviewer or Editor roles for the same submission (dynamic SoD).	All submission and revision actions are logged to support auditability and academic integrity.
U2 – Reviewer	Access only assigned submissions, submit reviews and recommendations. No permission to edit manuscripts or view author identities.	Reviewer must not activate Author or Editor roles for the same submission.	Review activities and deadlines are logged to ensure transparency and compliance with peer-review policies.
U3 – Editor	Assign reviewers, manage the editorial workflow, and record editorial decisions. No system configuration privileges.	Editor must not activate Author or Reviewer roles for submissions under editorial control.	Editorial assignments and decisions are auditable to ensure accountability and traceability.
U4 – Journal Manager	Manage journal configuration, sections, user roles, and subscriptions. No access to academic decision functions.	Administrative configuration activities are separated from editorial and review responsibilities.	All configuration changes, role assignments, and subscription updates are logged for governance and regulatory compliance.
U5 – Copyeditor	Responsible for linguistic and stylistic corrections on accepted manuscripts.	Permissions are enabled only after an editorial acceptance decision has been recorded.	Copyediting actions are logged; no access to reviewer identities or editorial decisions.
U6 – Layout Editor	Produce final presentation formats after copyediting completion.	Permissions are enabled only after the copyediting stage has been completed.	Layout activities are logged; no permission to modify scholarly content or access review data.
U7 – Proofreader	Perform final verification of typeset articles prior to publication.	May approve the final version but must not modify content or participate in earlier stages; dynamic SoD with Layout Editor may be enforced.	Proofreading approvals are logged to support publication integrity.

semantic foundation for analyzing the interaction of composed concerns during execution. This paper focuses on extending the UDML modeling framework to address these limitations.

2.3. Event-B and Rodin for Formal Verification

Event-B is a state-based formal modeling method grounded in first-order logic and typed set theory [9, 11]. It distinguishes static aspects, captured in contexts, from dynamic aspects, captured in machines comprising state variables, invariants, and guarded events. System behavior is defined through events, while correctness is ensured by proving that all events preserve specified invariants.

The Rodin platform [10] provides automated support for Event-B modeling and verification through the generation and discharge of proof obligations (POs). These POs ensure properties such as invariant preservation, event feasibility, and consistency across refinements. Event-B has been successfully applied to the verification of access-control policies and security constraints [12, 18], making it a suitable formal foundation for verifying RBAC-enriched domain models.

2.4. Research Questions

This study aims to establish a well-defined semantic foundation for unified executable domain models in DDD, with particular

emphasis on the formal incorporation of cross-cutting concerns into the domain model. We investigate the following research questions:
RQ1: How can formal operational semantics be defined for unified domain models in UDML?
RQ2: How can dynamic cross-cutting concerns, exemplified by RBAC, be specified as annotation-based DSLs and incorporated into unified domain models?

3. Overview of Our Approach

This section presents an overview of our approach to composing domain concerns and providing executable and verifiable semantics for unified domain models. As illustrated in Figure 2, the proposed method, named UDMM (Unified Domain Modeling Method), is organized as a three-step process. It takes as input descriptions of concern domains and produces, as output, verified unified executable domain models expressed in UDML.
Step 1: Metamodeling and concern composition. Given the description of a target concern domain, this step aims to specify and compose concerns, each of which is captured by a corresponding DSL_c, within the UDML framework. Building on the UDML concern composition mechanism [8], the UDML core is defined as a metamodel that provides explicit constructs for representing and composing multiple concern-specific DSLs through the metaconcepts `DomainModel`, `Annotable`, `Concern`, and `Annotation`. Specifically, a `DomainModel` consists of `Annotable` elements and `Concerns`. A `Concern` consists of `Annotations`. Each `Annotable` element is an instance of a metaclass from the UML metamodel for class diagrams, including `Package`, `Class`, `Property`, `Association`, and `Operation`.

Within this framework, typical domain concerns are captured using dedicated DSLs: DCSL specifies structural constraints, AGL defines executable behavioral models, and RBACDom is introduced as a concern-specific, annotation-based DSL for RBAC. Each DSL formalizes its corresponding concern at the modeling level and is incoporated into UDML via annotations attached to `Annotable` elements.

Step 2: Semantic definition via formal transformation. This step defines a formal semantics for the unified domain model. A transformation is defined from the UDML model enriched with DSL_c (including DCSL, AGL, and RBACDom) into Event-B, thereby providing a precise mathematical interpretation of the unified model.

In this transformation, the formal semantics of each DSL_c is defined based on a transition system. Behavioral specifications expressed in AGL determine the admissible state transitions, while structural constraints from DCSL and authorization constraints from RBACDom are interpreted as invariants and guards that restrict these transitions.

Step 3: Formal verification. This step performs formal verification using the Rodin platform [10]. Rodin automatically generates proof obligations corresponding to the unified executable semantics enriched with composed concerns. Discharging these proof obligations establishes the correctness of the composed domain model, including cross-concern consistency and the preservation of critical safety and security properties.

4. A Semantics Framework for Unified Domain Models

This section introduces a semantic framework for domain models, in which multiple concerns, including structural, behavioral, and security concerns, can be incorporated within the UDML framework.

4.1. Representing Unified Domain Models

Figure 3 presents the simplified metamodel of UDML for representing unified domain models.

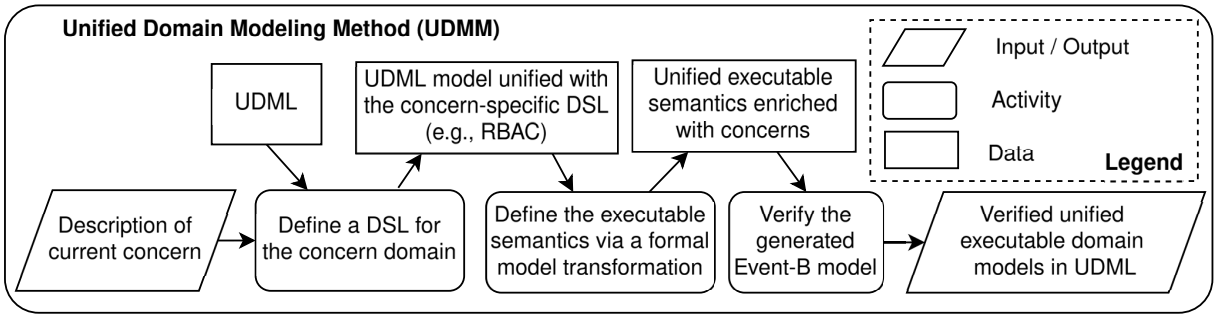

Figure 2. Overview of the proposed approach.

The metamodel is deliberately designed to support a precise and well-defined interpretation of unified domain models, in which structural, behavioral, and security concerns can be incorporated.

UDML provides a minimal yet expressive core—centred on the concepts `DomainModel`, `Concern`, `Annotable`, and `Annotation`—that serves as a neutral composition layer for heterogeneous concerns. This representation explicitly supports the systematic composition of domain concerns into a unified domain model.

Within this language-based composition approach, each DSL_c, such as DCSL [14] for structural constraints, AGL [8] for behavioral modelling as explained in Section 4.1.2, is defined by its own metamodel and remains conceptually independent. UDML adopts a metamodel-driven composition approach in which concerns are integrated through explicit correspondences mediated by the UDML core. This makes cross-concern interactions explicit, analyzable, and amenable to formal semantic interpretation. In particular, each concern-specific domain model (e.g., `DomainModelDcsl`, `DomainModelAgl`, and `DomainModelRbacDom`) are associated with a common `DomainModel` instance, ensuring a shared modeling context.

4.1.1. Incorporating Structural Constraints

To represent structural constraints and incorporate them into the unified domain model, we defined DCSL in our previous work [14]. Based on the metamodel of DCSL, we define its formal semantics to enable its incorporation into the unified domain model.

Abstract syntax. Figure 4 presents the DCSL metamodel, in our previous work [14], and illustrates the mapping between DCSL and UDML. The DCSL metamodel introduces the core metaconcepts `DomainModelDcsl`, `ConcernDcsl`, and `AnnotationDcsl`, which enable structural constraints to be specified and composed with the UDML core. Concrete structural constraints are expressed through annotation types `DClass`, `DAttr`, `DAssoc`, and `DOpt`, which are mapped respectively to `Class`, `Property`, `Property`, and `Operation`. To ensure the correctness of these mappings, well-formedness rules are specified using OCL constraints attached to the DCSL metamodel.

Semantics. DCSL is treated as a concern-specific, annotation-based DSL that captures the structural aspect of a domain model. Rather than introducing an independent structural metamodel, DCSL contributes a set of structural annotations andW well-formedness constraints that are bound to core UDML elements. We define the formal semantics of DCSL as follows.

- *Annotation-to-core binding:* Let the following carrier sets be given. $AN_{dcsl} \subseteq AN$ DCL, DAT, DAS, $DOP \subseteq AN_{dcsl}$, where DCL denotes instances of `DClass`; DAT denotes instances of `DAttr`; DAS denotes instances of `DAssoc`; DOP denotes instances of `DOpt`.

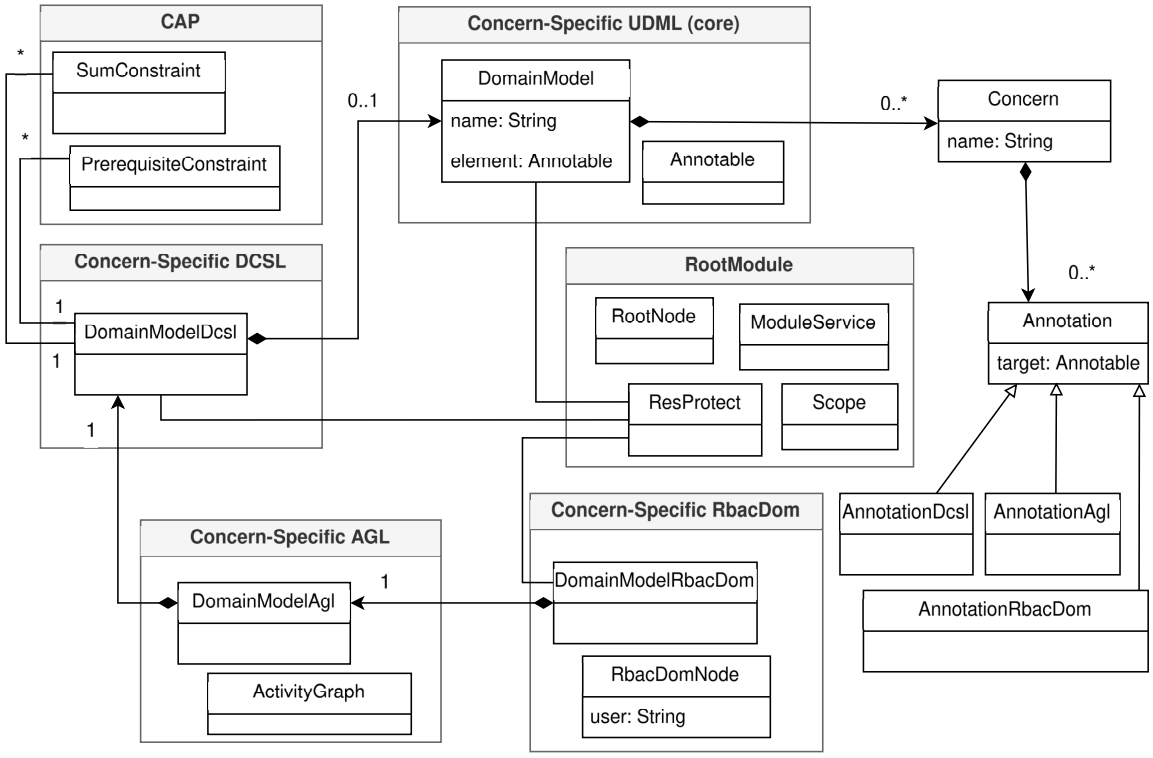

Figure 3. A simplified metamodel of UDML.

DCSL structural annotations are bound to UDML core elements through the UDML function $target : AN \rightarrow AE$, subject to the following typing constraints:
(D1) $\forall a \in DCL \cdot target(a) \in Class$;
(D2) $\forall a \in DAT \cdot target(a) \in Property$;
(D3) $\forall a \in DAS \cdot target(a) \in Property$;
(D4) $\forall a \in DOP \cdot target(a) \in Operation$.
These constraints formalise the intended mapping of DCSL metaconcepts to UDML core metaconcepts, ensuring that structural concern information is attached to the appropriate structural elements.

- *Structural well-formedness constraints:* Structural well-formedness constraints are defined to ensure the consistency of domain structure.
  Let R_{dcsl} denote the set of structural constraints contributed by DCSL (e.g., state-space constraints over domain fields and association ends).
  The structural concern model is well-formed iff all constraints in R_{dcsl} hold for the corresponding targets in the UDML model: $\bigwedge_{\varphi \in R_{dcsl}} \varphi$.
  These constraints constitute the structural correctness conditions that must be preserved by executions of the unified model.

4.1.2. Incorporating Domain Behaviour

To represent domain behaviors and integrate them into the unified domain model, we further refine AGL by constructing its metamodel and defining its formal semantics.

Abstract syntax. Figure 5 depicts the AGL metamodel, capturing executable domain behavior via activity graphs, nodes, and control-flow relations. We define it [17] within unified executable domain models to support semantic integration with structural and security concerns.

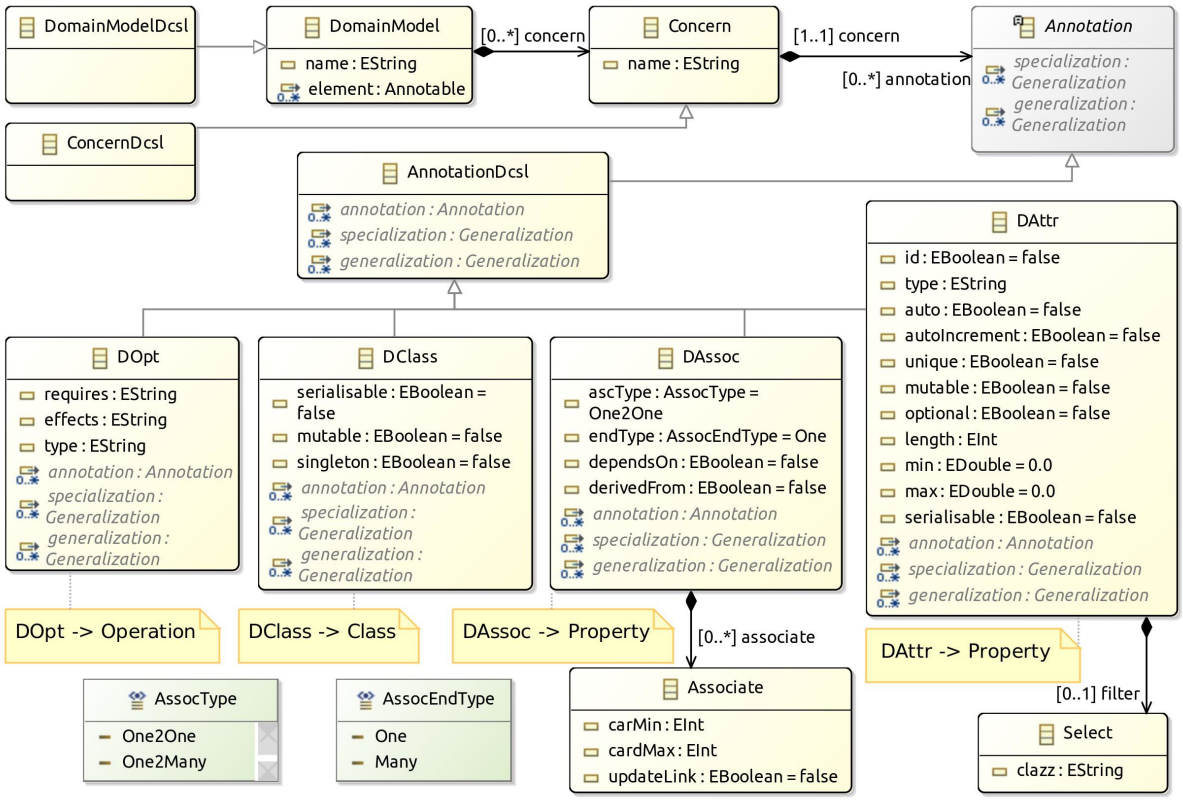

Figure 4. The DCSL metamodel to capture the concern domain.

The core metaconcepts of AGL are as follows. `ActivityGraph` represents a unit of executable domain behaviour and organises behaviour as a directed graph. It is associated with a `RootNode`, which identifies the entry point used to activate the corresponding behavioral flow at execution time. `Node` represents an atomic unit of behaviour and defines an explicit execution boundary. Each node references a domain class via `refCls`, thereby linking behavioral execution to the structural domain model. `Edge` defines admissible control-flow relations between nodes and constrains the possible progression of execution. `ModuleAct` specifies the concrete actions executed at a node. Each `ModuleAct` is associated with a corresponding `ModuleService` defined in the `RootModule`, ensuring that behavioral actions are consistently realised within the modular execution architecture. `PreState` and `PostState` characterise the expected domain states before and after the execution of a module action, respectively.

In this metamodel, each `ActivityGraph` owns its nodes and edges, forming a well-scoped behavioral execution context. Behavioral execution is initiated at the `RootNode`, proceeds along admissible edges, and invokes module services through `ModuleAct` elements. By referencing domain classes at node level, AGL provides an explicit linkage between behaviour and the structural domain model, forming a foundation for executable semantics within UDML.

Semantics. The behavioral aspect of UDML models is defined in AGL through activity graphs

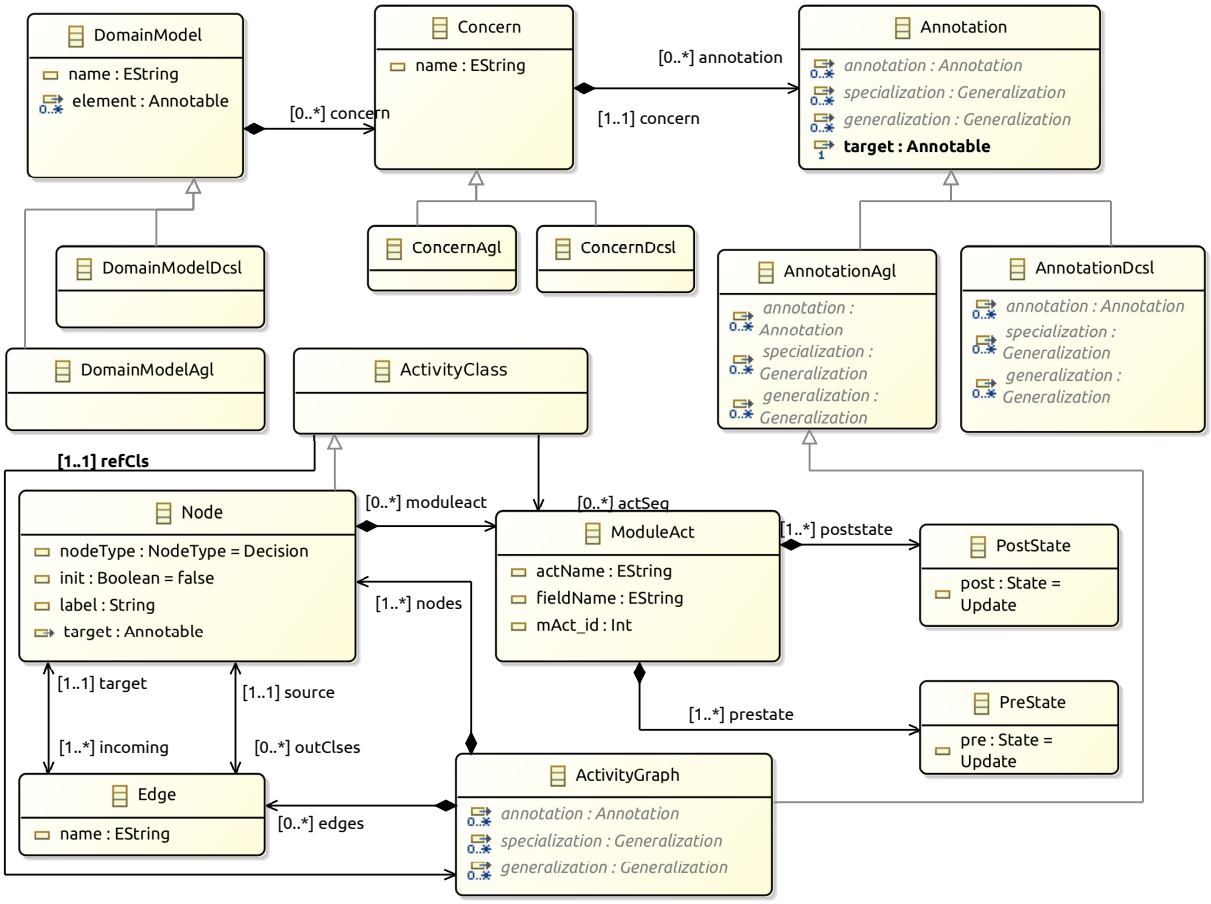

Figure 5. The AGL metamodel for capturing executable domain behavior.

incorporated with the structural domain model. We define the formal semantics of AGL as follows.

Let: AG, ND, ED, MA, AC denote the sets of activity graphs, activity nodes, edges, module actions, and activity classes, respectively.

- *Activity graph:* An activity graph $g \in AG$ is defined as a tuple.
  $g = \langle N_g, E_g, src_g, tgt_g, init_g, actSeq, refActCls \rangle$, where $N_g \subseteq ND$ is the finite set of activity nodes; $E_g \subseteq ED$ is the finite set of edges; $src_g, tgt_g : E_g \to N_g$ define the source and target of each edge; $init_g \in N_g$ is the initial node of the activity graph; $actSeq : ND \to \mathcal{P}(MA)$ associates each node with a (possibly empty) set of module actions executed at that node; $refActCls : ND \to AC$ associates each node with its referenced activity class.

- *Activity classes:* Activity classes act as behavioral extensions of domain classes, providing the structural context in which activity graphs are interpreted. Each activity graph is associated with exactly one activity class, and activity nodes refer to this class when executing module actions.

- *Annotable nodes:* A key design decision in UDML is that activity nodes are annotable elements. $ND \subseteq AE$. This enables concern-specific semantics, including security constraints specified in RBACDom, to be

attached directly to behavioral execution boundaries.

- *AGL well-formedness constraints:* The following well-formedness constraints must hold for any activity graph $g \in AG$.
  (A1) $N_g \neq \emptyset$;
  (A2) $\forall e \in E_g \cdot src_g(e) \in N_g \wedge tgt_g(e) \in N_g$;
  (A3) $init_g \in N_g$;
  (A4) $\forall n \in N_g \cdot refActCls(n)$ is defined.

In our previous work [17], an execution of an AGL model is defined as a valid sequence of enabled atomic actions that induces a corresponding sequence of reachable system states, following the control flow of the activity graph and respecting the pre- and post-state conditions of each action.

Definition 1 (Execution of an AGL model). *Let $\mathcal{G} = (N, E)$ be an AGL activity graph, where each node $n \in N$ is associated with a structured atomic action $SAA(n)$, denoting a set of admissible atomic action sequences (ASEs). An execution of $\mathcal{G}$ is a (finite or infinite) sequence of nodes $\pi = (n_0, n_1, \ldots, n_k)$ such that: (i) for all $i < k$, $(n_i, n_{i+1}) \in E$; and (ii) there exists a sequence of atomic action sequences $S_0, S_1, \ldots, S_k$, with $S_i \in SAA(n_i)$, whose induced state transformations are compatible, i.e., the post-state of S_i satisfies the pre-state of S_{i+1}. Equivalently, an execution corresponds to a valid atomic action sequence that induces a sequence of reachable system states.*

4.1.3. Incorporating Security Concerns

To support the explicit modeling of access control, we introduce RBACDom as a concern-specific DSL dedicated to RBAC within unified domain models. RBACDom is designed in accordance with the concern-oriented principles of UDML, allowing security policies to be specified independently while remaining composable with structural and behavioral concerns. In line with the UDML language design, RBACDom is defined in terms of its abstract syntax, concrete syntax, and formal semantics, enabling authorization constraints to be systematically incorporated into unified domain models.

Abstract syntax. Figure 6 shows the abstract syntax of RBACDom for capturing security concerns. RBACDom is defined by a dedicated metamodel based on UML metaconcepts [19]. This metamodel captures the core RBAC concepts and their structural relationships, and specifies how access control information is represented and associated with UDML models.

RBACDom is specified as an independent concern-specific DSL whose domain model, `DomainModelRbacDom`, is composed with other concern-specific domain models through the UDML core. In particular, `DomainModelRbacDom` is associated with `DomainModelAgl` at the UDML level, preserving a clear separation between behavioral modeling and access control specification.

The metaconcept `DomainModelRbacDom` defines an attribute *combiningAlg* that specifies the rule-combining strategy associated with the RBACDom model. This attribute determines how multiple applicable annotations attached to the same behavioral node are evaluated (e.g., denyOverrides, permitOverrides, firstApplicable).

Node-level integration is realised through the metaconcept `RbacDomNode`. An instance of `RbacDomNode` represents an RBAC policy specification and is declared within an `ActivityGraph` as part of `DomainModelRbacDom`. Each `RbacDomNode` establishes an explicit correspondence to a behavioral node defined in AGL via a reference attribute (e.g., `aglNode`), which identifies the target node by its label. This correspondence is structural in nature and does not modify the abstract syntax of `AGL :: Node`.

The metaconcept `RbacDomNode` captures RBAC-specific properties, including required

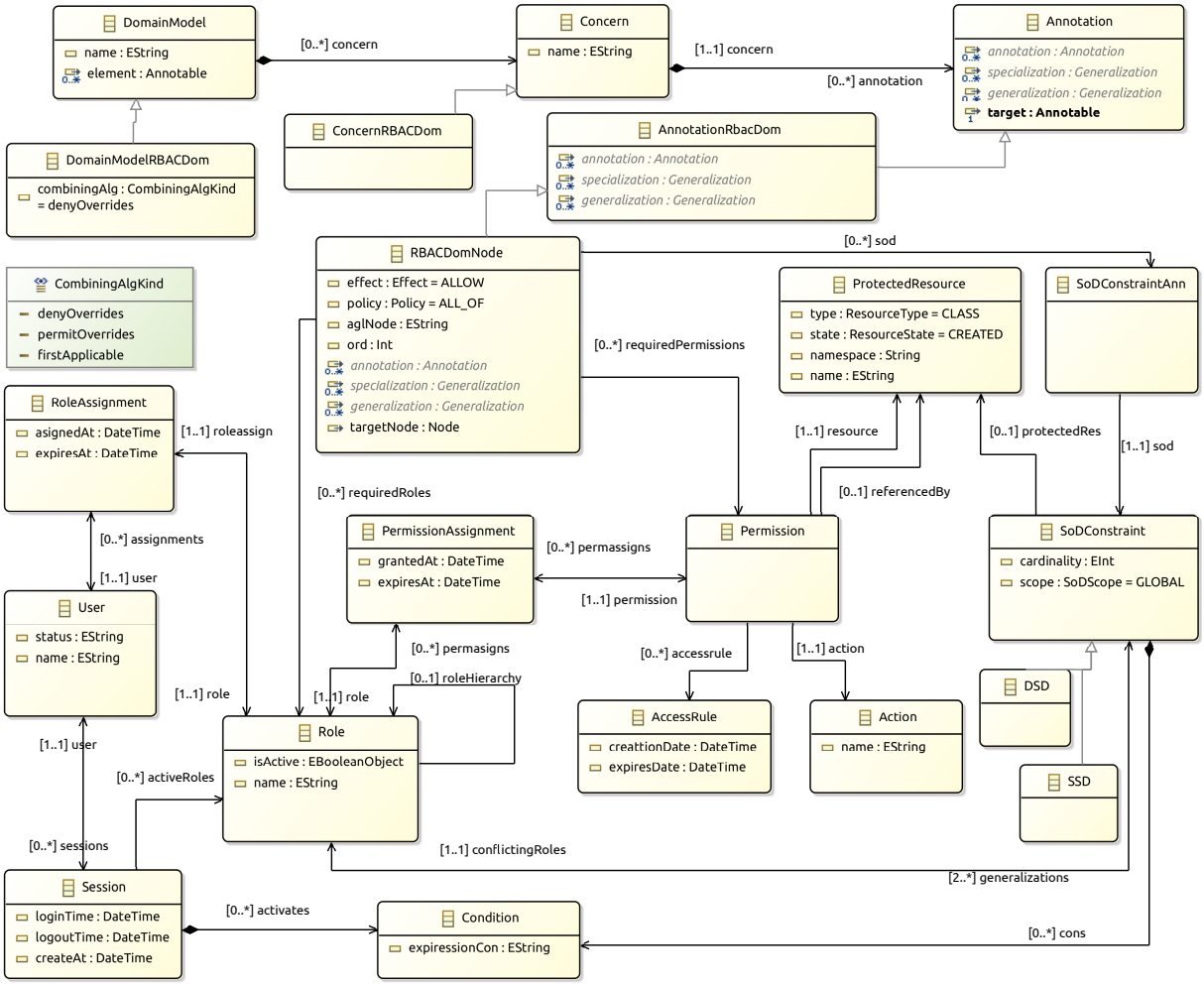

Figure 6. The RBACDom metamodel captures the security concern.

roles and/or permissions, policy evaluation mode (`ALL_OF` or `ANY_OF`), effect (`ALLOW` or `DENY`), and optional separation-of-duty constraints. An optional scope predicate may be used to restrict the applicability of a policy to a subset of domain objects, thereby supporting contextual access control at the modeling level. In addition, `RbacDomNode` defines an optional attribute *ord*, representing the evaluation order of annotations. This attribute is used only when the combining strategy is firstApplicable, where annotations are evaluated according to their assigned order.

Configurable rule-combining parameters. In practical access-control frameworks, multiple authorization rules may apply simultaneously to the same protected operation. Different policy systems therefore adopt different rule-combining strategies to determine the final authorization decision. To capture this flexibility at the modeling level, RBACDom provides an explicit parameter that specifies how multiple applicable annotations are combined. Let $CA = \{\text{denyOverrides}, \text{permitOverrides}, \text{firstApplicable}\}$ be the set of admissible rule-combining strategies. The attribute $combiningAlg : DomainModelRbacDom \rightarrow CA$ associates a combining strategy with each RBACDom model. If no value is explicitly specified, the default

strategy denyOverrides is assumed.

Let $R \in DomainModelRbacDom$ denote the RBACDom model. For order-sensitive evaluation under the firstApplicable strategy, we additionally consider a partial function $ord : RbacDomNode \rightharpoonup \mathbb{N}$, where $ord(rna)$ denotes the evaluation order of annotation rna.

Let $Ann(n) \subseteq RbacDomNode$ denote the set of RBACDom annotations associated with node $n \in ND$. The following well-formedness condition ensures deterministic evaluation of node-local policies: $(RC1)$ $combiningAlg(R) = \text{firstApplicable} \Rightarrow \forall n \in ND \cdot \forall rna_1, rna_2 \in Ann(n) \cdot (ord(rna_1) = ord(rna_2) \Rightarrow rna_1 = rna_2)$.

RBACDom further defines the standard RBAC concepts `User`, `Role`, `Permission`, and `Session` independently of UDML domain elements. User–role and role–permission assignments are represented explicitly as relations, while permissions are modeled as ⟨action, protected resource⟩ pairs. Protected resources may optionally be bound to UDML annotable elements to support integration with domain models.

To represent constraint for RBACDom, introduces the abstract metaconcept `SoDConstraint`, with specializations for static `SSD` and dynamic `DSD` separation of duty. Explicit modeling of SoD constraints improves modularity and structural analyzability of access control specifications.

The RBACDom metamodel is equipped with a set of structural well-formedness rules, summarised in Table 2. These rules ensure the internal consistency of RBAC specifications and the validity of their attachment to UDML models prior to semantic interpretation or formal verification.

Concrete syntax. To support practical modeling and tool integration, we define an annotation-based textual concrete syntax for RBACDom that is consistent with the concern-oriented, metamodel-driven composition principles of UDML. The concrete syntax is derived from the abstract syntax metamodel of RBACDom by defining a corresponding concrete syntax metamodel suitable for embedding in a host object-oriented programming language (e.g., Java).

In line with UDML's annotation-driven integration style, RBACDom is realized as an external annotation-based DSL. RBAC policies are specified textually as annotations that instantiate `RbacDomNode` elements of the RBACDom metamodel, while their semantic association with executable behaviour is established through explicit node-level correspondence to AGL nodes. This design preserves the separation between behavioral and security concerns while enabling their systematic composition at the modeling level.

The concrete syntax is centred around the annotation `@RBACDomNode`, which represents a node-level RBACDom specification. Each `@RBACDomNode` instance corresponds to an element of the abstract syntax `RbacDomNode` and defines authorization constraints associated with a behavioral execution boundary. The annotation body encodes RBAC-specific properties such as required roles or permissions, policy evaluation mode, effect, and optional separation-of-duty constraints.

The following listing illustrates the annotation-based textual syntax used to specify RBACDom policies:

```
@RBACDomNode(
    id     = "U.ID",
    roles  = {<RoleName>},
    perms  = {(<Op>, <Res>), (<...>)},
    policy = ALL_OF | ANY_OF,
    effect = ALLOW | DENY,
    scope  = {<ScopePredicate>,<...>},
    sod    = { SSD | DSD | ... },
    ord    = <Integer>
)
```

This notation captures authorization requirements at the level of executable behaviour.

Table 2. Structural well-formedness rules of the RBACDom metamodel

Constraint	Description
WF-S1: Unique RBAC identifiers	All RBAC entities (users, roles, permissions, actions, and protected resources) must have unique identifiers within a RBACDom model.
WF-S2: Well-typed role assignments	Each user–role assignment must reference an existing user and an existing role. Dangling or undefined role assignments are not permitted.
WF-S3: Well-typed permission assignments	Each role–permission assignment must reference an existing role and a valid permission defined in the model.
WF-S4: Valid permission definition	Each permission must be defined as exactly one action applied to exactly one protected resource, following the RBAC interpretation of permissions as $\langle operation, object \rangle$ pairs.
WF-S5: Unique permission tuples	No two permissions may define the same action–resource pair.
WF-S6: Optional resource binding	A protected resource may optionally be bound to a concrete UDML domain element. Unbound resources are permitted to support late binding.
WF-S7: Resource–domain type compatibility	If a protected resource is bound to a domain element, the resource type must be compatible with the kind of UDML element to which it is bound.
WF-S8: Well-formed separation of duty constraints	Each SoD constraint must reference at least two distinct roles and define a valid cardinality ($n \geq 2$).
WF-S9: Static separation of duty consistency	Static SoD constraints must not be violated by user–role assignments.
WF-S10: Annotation attachment and scope	All RBAC annotations must be attached to well-defined UDML elements.
WF-S11: Deterministic ordering for firstApplicable	If *combiningAlg* = firstApplicable, all `RbacDomNode` annotations attached to the same behavioral node must have distinct evaluation orders.

Role and permission requirements specify who may execute a behavioral node, policy operators define how1 multiple requirements are combined, and the effect determines whether the policy permits or denies execution. Optional scope predicates and SoD constraints further restrict applicability without introducing additional behavioral constructs.

Example. The following example illustrates an RBACDom specification for the `Author` role in the OJS domain:

```
@RBACDomNode(
    id     = "U1.Author",
    roles  = {"Author"},
    perms  = {
    ("CreateSubmission", "Submission"),
    ("UploadManuscriptFile",
     "ManuscriptFile")
    },
    policy = ANY_OF,
    effect = ALLOW,
    sod    = {DSD},
    ord    = 1
)
```

The annotation is declared as part of the RBACDom model and is associated with a behavioral node through an explicit correspondence to the node label defined in the AGL activity graph.

4.1.4. Mapping between AGL and RBACDom

Security concerns are captured independently in RBACDom via the metaconcept `RbacDomNode`, which specifies authorization constraints, role requirements, and related security properties. Rather than embedding security directly into behavioral nodes, RBACDom nodes are composed with AGL nodes through an explicit correspondence relation—typically realized via a reference or label (e.g., `aglNode`)—that identifies the behavioral node whose execution is constrained.

This mapping preserves the modularity of concern-specific DSLs while aligning their semantics within UDML. AGL defines admissible behavioral transitions, whereas RBACDom constrains executability by restricting transition activation under authorization conditions. These concerns are unified through the UDML core over a shared behavioral interpretation, without conflating their abstract syntaxes. Executable realization is supported via the *RootModule*, which encapsulates composed DCSL, AGL, and RBACDom specifications into a coherent unit. Governed by `ModuleService` and mediated by `ResProtect` and scope definitions, this structure ensures consistent enforcement of behavior and access control, yielding a unified abstract syntax that underpins executable semantics and formal verification.

The following excerpt illustrates how RBACDom specifications coexist with behavioral definitions in an annotation-based host language. Behavioral nodes are defined using AGL annotations, while RBACDom nodes are declared independently and linked to behavioral nodes through correspondence relations rather than direct syntactic embedding.

Example. The following example illustrates how RBACDom annotations are used to specify authorization constraints for the *Submission* activity in the OJS domain, under the `firstApplicable` rule-combining strategy.

```
@AGraph(nodes = {
  @ANode(label = "Submission",
    nodeType = NodeType.Action,
    init = true,
    refCls = Submission.class,
    outNodes = {"SubmissionQueue"},
    moduleact = {@MAct(
      actName = ActNames.newObject,
      poststate = {State.Created})}
  )
  @RBACDomNode(
    id = "U1.GuestDeny",
    aglNode = "Submission",
    effect  = Effect.DENY,
    roles   = {Role.Guest},
    ord     = 1
  ),
  @RBACDomNode(
    id = "U1.AuthorAllow",
    aglNode = "Submission",
    effect  = Effect.ALLOW,
    roles   = {Role.Author},
    perms   = {
("CreateSubmission","Submission"),
("UploadManuscriptFile","ManuscriptFile")
     },
    policy  = Policy.ANY_OF,
    sod     = {DSD},
    ord     = 2
  ), ...
}
)
  public class SubmissionMng{
      ...
  }
```

When the combining strategy is `firstApplicable`, annotations are evaluated according to the `ord` attribute. In this example, the first applicable rule determines the final decision. If the user satisfies the first rule (e.g., has role `Guest`), access to the *Submission* activity is denied. Otherwise, the second rule is evaluated, allowing execution for users with the `Author` role who satisfy the required permissions.

Semantics for RBACDom. Security aspects are captured by RBACDom, a concern-specific, annotation-based DSL that formalises RBAC together with SoD constraints and binds them to annotable UDML elements, in particular AGL activity nodes. We define the formal semantics of AGL as follows.

Let the following carrier sets be given: USR, R, P, SES, ACT, RES, SC, $RbacDomNode$ denoting users, roles, permissions, sessions, actions, protected resources, SoD constraints, and node RBAC annotations, respectively.

- *Role and permission assignment:* RBACDom captures assignments as relations. $UA \subseteq USR \times R$; $PA \subseteq R \times P$, where $(u, r) \in UA$ means user u is assigned role r, and $(r, p) \in PA$ means role r is granted permission p.SoD

- *Sessions and active roles:* Sessions are related to users and activated roles by. $userOf : SES \rightarrow USR$ $activeRoles \subseteq SES \times R$, where $(s, r) \in activeRoles$ means role r is active in session s.

- *Permissions and protected resources:* Each permission is characterised by an action and a protected resource. $permAction : P \rightarrow ACT$; $permRes : P \rightarrow SES$. Protected resources may be bound to UDML annotable elements to enable domain-level referencing: $bindsTo : RES \rightharpoonup AE$

- *Separation of duty (SoD):* A SoD constraint $c \in SC$ is defined as a tuple. $c = \langle conflictSet(c), card(c), kind(c), scope(c)\rangle$, where $conflictSet(c) \subseteq R$, $card(c) \in \mathbb{N}$, $kind(c) \in \{SSD, DSD\}$, and $scope(c)$ denotes the constraint scope (e.g., *GLOBAL*).

- *Node-anchored RBACDom annotations:* RBACDom policies are attached to behavioral execution boundaries through node-level annotations. Let $ND \subseteq AE$ be the set of annotable activity nodes defined by AGL. Each RBACDom annotation $rna \in RbacDomNode$ is attached to exactly one activity node via the function: $targetNode : RbacDomNode \rightarrow ND$. Each annotation provides the following abstract syntax components: $policy : RbacDomNode \rightarrow \{ALL\_OF, ANY\_OF\}$; $effect : RbacDomNode \rightarrow \{ALLOW, DENY\}$; $ReqRoles : RbacDomNode \rightarrow \mathcal{P}(R)$; $ReqPerms : RbacDomNode \rightarrow \mathcal{P}(P)$; $SoD : RbacDomNode \rightarrow \mathcal{P}(SC)$

Mapping for RBACDom. In UDML, RBACDom is interpreted as a security concern that constrains behavioral executability through node-level correspondences with AGL activity nodes. Each `RbacDomNode` induces an authorization condition on its target node, restricting node enablement without modifying the behavioral structure or control flow. An activity node is therefore executable only if it is enabled by AGL and all associated RBACDom constraints are satisfied in the current authorization context.

In general, multiple RBACDom annotations may be associated with the same behavioral node. The final authorization decision therefore depends on the rule-combining strategy selected by the RBACDom model. As introduced in Section 4.1.3, the function $combiningAlg : DomainModelRbacDom \rightarrow CA$ specifies how multiple applicable annotations are evaluated. The semantics defined below therefore parameterize authorization evaluation with respect to this rule-combining strategy. Under this semantics, AGL defines behavioral evolution while RBACDom prunes inadmissible transitions, providing a compositional basis for semantics-preserving transformation and formal verification.

Definition 2 (RBACDom Node Correspondence). *Let $G = (N, E)$ be an AGL activity graph with $N \subseteq ND$. Node-level integration of RBACDom into G is defined by the total mapping $targetNode : RbacDomNode \rightarrow ND$ which binds each $rna \in RbacDomNode$ to its unique target node. A node $n \in N$ is enabled in an execution context only if all RBAC and separation-of-duty constraints specified by every rna with $targetNode(rna) = n$ are satisfied.*

Node-Enablement semantics for RBACDom-annotated graphs. This subsection defines

the formal node-enablement semantics for RBACDom-annotated activity graphs. Authorization is interpreted as a semantic constraint on behavioral execution: an activity node may be enabled and executed in a given session only if its associated RBAC policies are satisfied. The semantics is defined at the level of activity graph nodes, which serve as execution boundaries for domain behaviour.

A node n is enabled in a snapshot $\sigma \in \Sigma$ if and only if it is enabled by the AGL control-flow semantics and all RBACDom constraints are satisfied with respect to the authorization context contained in $S_{DSL_c}(\sigma)$.

Auxiliary functions and typing. The auxiliary functions and predicates used in the enablement semantics are defined and typed as follows:
$Ann : ND \to \mathcal{P}(RbacDomNode)$
associates each activity node with its set of RBACDom annotations.

For each annotation $rna \in RbacDomNode$, the following mapping functions are defined as following.

$$policy : RbacDomNode \to \{ALL\_OF, ANY\_OF\}$$
$$effect : RbacDomNode \to \{ALLOW, DENY\}$$
$$ReqRoles : RbacDomNode \to \mathcal{P}(R)$$
$$ReqPerms : RbacDomNode \to \mathcal{P}(P)$$
$$SoD : RbacDomNode \to \mathcal{P}(SC)$$

The permissions enabled by a set of roles are derived by: $Perms : \mathcal{P}(R) \to \mathcal{P}(P)$, where $Perms(R') = \{p \in P \mid \exists r \in R' \cdot (r, p) \in PA\}$

The predicate $Violates : SC \times SES \to \mathbb{B}$ indicates whether a separation-of-duty constraint is violated in a given session.

A node-anchored RBACDom annotation $rna \in Ann(n)$ is *applicable* in a session $s \in S$ if and only if all its role, permission, and SoD conditions are satisfied:
$Applies(rna, s) \quad s \in SES\,RolesSat(rna, s) \wedge PermsSat(rna, s) \wedge SoDSat(rna, s)$

The component predicates are defined as:

$$RolesSat(rna, s) \triangleq (policy(rna) = ALL\_OF \Rightarrow ReqRoles(rna) \subseteq activeRoles(s)) \wedge (policy(rna) = ANY\_OF \Rightarrow ReqRoles(rna) \cap activeRoles(s) \neq \emptyset)$$
$$PermsSat(rna, s) \triangleq ReqPerms(rna) \subseteq Perms(activeRoles(s))$$
$$SoDSat(rna, s) \triangleq \forall c \in SoD(rna) \cdot \neg Violates(c, s)$$

Rule-combining semantics. Let $s \in SES$ denote a session state, where SES is the set of session states. For a behavioral node $n \in ND$, an RBACDom annotation $rna \in RbacDomNode$ is said to be applicable in state s if its role, permission, and contextual constraints are satisfied. We denote this predicate by $Applies(rna, s)$. The set of applicable RBACDom annotations associated with node n in state s is defined as $App(n, s) = \{rna \in Ann(n) \mid Applies(rna, s)\}$. Let $effect : RbacDomNode \to \{ALLOW, DENY\}$ denote the authorization effect associated with an RBACDom annotation. The authorization decision for node n in state s is determined by a strategy-dependent decision function
$Dec : ND \times SES \to \{ALLOW, DENY\}$. The definition of $Dec(n, s)$ depends on the rule-combining strategy selected by $combiningAlg(R)$.

Deny-Overrides. If $combiningAlg(R) =$ denyOverrides, the decision is $Dec(n, s) =$
$$\begin{cases} DENY & \text{if } \exists rna \in App(n, s) \cdot effect(rna) = DENY \\ ALLOW & \text{if } \exists rna \in App(n, s) \cdot effect(rna) = ALLOW \\ DENY & \text{otherwise.} \end{cases}$$

Permit-Overrides. If $combiningAlg(R) =$ permitOverrides, the decision is $Dec(n, s) =$
$$\begin{cases} ALLOW & \text{if } \exists rna \in App(n, s) \cdot effect(rna) = ALLOW \\ DENY & \text{if } \neg\exists rna \in App(n, s) \cdot effect(rna) = ALLOW \\ & \quad \wedge \exists rna \in App(n, s) \cdot effect(rna) = DENY \\ DENY & \text{otherwise.} \end{cases}$$

First-Applicable. If $combiningAlg(R) =$ firstApplicable, rules are evaluated according to

the ordering relation ord.

$$first(n, s) = \arg \min_{rna \in App(n,s)} ord(rna)$$

The decision becomes $Dec(n, s) = \begin{cases} effect(first(n, s)) & \text{if } App(n, s) \neq \emptyset \\ DENY & \text{otherwise.} \end{cases}$

Finally, a behavioral node is considered authorization-enabled if $Enable_{RBAC}(n, s) \triangleq Dec(n, s) = ALLOW$.

Thus, the RBACDom component of node enablement is captured by $Enable_{RBAC}(n, s)$. A behavioral node n is executable only if it is enabled by the behavioral semantics of AGL and the authorization condition $Enable_{RBAC}(n, s)$ holds.

4.2. Formal Semantics of Domain Models

This section provides a semantic framework for unified domain models in UDML. A unified domain model is interpreted as the semantic integration of a core domain model with a set of composed concern-specific DSLs. The semantics is defined in terms of executions, where each execution corresponds to a sequence of system snapshots representing the evolution of the model over time.

Each system snapshot captures a system state induced by the unified model and is characterized by the combination of a core model snapshots and the snapshots contributed by the composed concern models, formally represented as $(S_{\text{core}}, S_{DSL_c})$. On this basis, the operational semantics of a unified domain model is defined by the admissible execution paths along which such snapshots evolve, establishing a semantic contract at the UDML level without yet committing to the interpretation of individual concerns.

4.2.1. Formal Definition of UDML Models.

The formal definition of UDML models as unified executable domain models obtained by composing multiple orthogonal but semantically interrelated concerns.

Definition 3 (UDML Model). *A UDML model is defined as a tuple:* $\mathcal{M} = \langle \mathcal{U}, \mathcal{S}, \mathcal{B}, \mathcal{R}, C \rangle$ *where:*

- $\mathcal{U}$ *is the UDML core composition metamodel, providing the abstractions for concern modularisation and annotation binding, while remaining agnostic of operational semantics;*
- $\mathcal{S}$ *is the DCSL structural concern model, consisting of DCSL annotations and constraints bound to structural elements of the domain model;*
- $\mathcal{B}$ *is the AGL behavioral concern model, consisting of activity graphs whose nodes and edges characterise admissible behavioral evolution and execution boundaries for domain actions;*
- $\mathcal{R}$ *is the RBACDom security concern model, specifying authorization and SoD constraints that restrict the executability of behavioral elements (in particular, node execution);*
- C *is the set of global constraints, including well-formedness conditions and cross-concern consistency constraints that must be preserved by all model executions.*

This decomposition clarifies the role of each modeling concern while enabling their systematic integration within a single executable domain model. In particular, the behavioral model $\mathcal{B}$ determines the space of admissible state transitions, the security model $\mathcal{R}$ constrains which transitions are executable under given authorization conditions, and the constraint set C defines global correctness properties that must be maintained throughout execution.

Formalization of the UDML Core. The UDML core provides the foundational abstractions for composing multiple concern-specific DSLs in a uniform and non-invasive manner. It defines

the structural glue that enables the integration of structural, behavioral, and security concerns without prescribing executable semantics by itself. We define the formal semantics of UDML as follows.

- *Core domain elements:* The UDML core is formalised as the tuple
  $\mathcal{U} = \langle DM, CN, AN, AE, elements, concerns, annotations, target \rangle$
  Let the following pairwise disjoint carrier sets be given: DM, CN, AN, AE, where DM denotes the set of domain models; CN denotes the set of concerns associated with domain models; AN denotes the set of annotations used to represent concern-specific information; AE denotes the set of annotable elements, i.e., domain model elements that may be extended with annotations.

  Let the following core domain elements be given: $Class$, $Property$, $Operation$, $Association \subseteq AE$. These elements constitute the abstract syntax of the domain model in UDML. They represent the fundamental structural modeling concepts—analogous to UML classes, properties, operations, and associations—over which domain behaviour and concern-specific semantics are defined. Annotable elements thus form a common target space for concern integration: all concern-specific DSLs, such as DCSL, AGL, and RBACDom, contribute semantics by attaching annotations to elements in AE rather than by redefining the domain model structure itself.

- *Core relations:* The core relations of $\mathcal{U}$ are defined as follows.
  $elements \subseteq DM \times AE$ associates each domain model with its annotable elements; $concerns \subseteq DM \times CN$ associates each domain model with its declared concerns; $annotations \subseteq CN \times AN$ associates each concern with its annotations; $target : AN \to AE$ maps each annotation to its target annotable element.

- *Well-formedness rules:* The following well-formedness constraints must hold.
  (U1) $\forall a \in AN \cdot target(a) \in AE$;
  (U2) $\forall a \in AN \cdot \exists! c \in CN : (c, a) \in annotations$;
  (U3) $\forall c \in CN \cdot \exists! d \in DM : (d, c) \in concerns$.
  (U4) $\forall a \in AN \cdot \exists d \in DM : (d, target(a)) \in elements \wedge \exists c \in CN : (d, c) \in concerns \wedge (c, a) \in annotations$.
  These constraints ensure a well-defined ownership and attachment structure for concerns and annotations within a domain model.

Definition 4 (System Snapshot / State). *Let $\mathcal{M}$ be a UDML model. The set Σ denotes the set of system snapshots induced by $\mathcal{M}$. Each snapshot $\sigma \in \Sigma$ is defined as a structured state:*

$$\sigma \triangleq \langle S_{core}(\sigma),\ S_{DSL_c}(\sigma) \rangle,$$

where $S_{core}(\sigma)$ captures the current state of the core domain model (e.g., valuations of structural elements and relations), and $S_{DSL_c}(\sigma)$ captures the states contributed by the composed concern-specific DSLs, including behavioral execution information (from AGL) and authorization context (from RBACDom). The alignment between these components is induced by the bindings defined in the UDML core.

Definition 5 (Valid Snapshot). *Let $\mathcal{M}$ be a UDML model and $\sigma \in \Sigma$ a snapshot. We say that σ is* valid*, written (σ), iff σ satisfies all structural constraints induced by DCSL, all security invariants induced by RBACDom, and all invariants induced by other composed concern-specific DSLs, together with the binding conditions defined in the UDML core that align $S_{core}(\sigma)$ and $S_{DSL_c}(\sigma)$.*

Definition 6 (Operational Semantics). *The semantics of a UDML model $\mathcal{M}$ is defined as a labelled transition system*

$$[\![\mathcal{M}]\!] = \langle \Sigma, \Sigma_0, \rightarrow \rangle,$$

where Σ is the set of snapshots, $\Sigma_0 \subseteq \Sigma$ is the set of initial snapshots, and $\rightarrow \subseteq \Sigma \times ND \times \Sigma$ is the transition relation.
An execution (or path) of $\mathcal{M}$ is a (finite or infinite) sequence $\sigma_0, \sigma_1, \ldots$ such that $\sigma_0 \in \Sigma_0$, (σ_0), and for each $i \geq 0$, $\sigma_i \xrightarrow{n_i} \sigma_{i+1}$ with (σ_{i+1}).

An execution represents a possible run of the model $\mathcal{M}$, starting from an initial snapshot and evolving through successive node executions. Each transition corresponds to the execution of a node that transforms the system state from one snapshot to the next. At each step, the node-enablement condition $Enable(n_i, s)$ is evaluated in the current snapshot σ_i, using the session component $s \in SES$ contained in S_{DSL_c}. The transition to σ_{i+1} is permitted only if it preserves all structural constraints induced by DCSL and all invariants contributed by the composed DSL$_c$s in the resulting snapshot.

In the following, we make explicit the mapping of AGL executions in snapshots to corresponding UDML snapshots.

4.2.2. Mapping Execution in AGL to UDML

Definition 7 (Snapshot mapping). *Let $\mathcal{M}$ be a UDML model and Σ_{AGL} the set of snapshots induced by its AGL specification. An AGL snapshot $q \in \Sigma_{AGL}$ corresponds to a UDML snapshot $\sigma \in \Sigma$, written $q \sim \sigma$, iff $q = \pi_{AGL}(\sigma)$, where $\pi_{AGL} : \Sigma \rightarrow \Sigma_{AGL}$ projects the AGL behavioral component from $S_{DSL_c}(\sigma)$.*

Example. The following example illustrates an execution of the OJS system as a sequence of snapshots induced by its AGL specification. $q_0 \xrightarrow{\text{submit}} q_1 \xrightarrow{\text{approve}} q_2$ corresponds to the UDML execution $\sigma_0 \xrightarrow{\text{submit}} \sigma_1 \xrightarrow{\text{approve}} \sigma_2$. We have the snapshot mappings $q_i \sim \sigma_i$, (i=0,2). If RBACDom requires role Manager for approve, the second step is enabled only when the authorization holds; Otherwise execution stops at σ_1. In both cases, behavioral correspondence is preserved.

Theorem 1 (Mapping). *Let $\mathcal{M}$ be a UDML model that composes AGL with a set of concern-specific DSLs. Assume that each composed concern DSL constrains behaviour only by restricting node enablement (i.e., by adding guards and/or state invariants), and does not modify the AGL snapshot. Then: (i) every UDML execution projects via π_{AGL} to a valid AGL execution; and (ii) every AGL execution that satisfies the enablement restrictions and invariants induced by the composed concerns lifts to a corresponding UDML execution.*

Proof **(Mapping one-by-one).**

> Let $\mathcal{M}$ be a well-formed UDML model. By construction, each UDML snapshot $\sigma \in \Sigma$ contains an AGL behavioral component, and the projection π_{AGL} establishes a mapping in UDML snapshots to AGL snapshot.

Soundness. Every UDML step $\sigma_i \xrightarrow{n_i} \sigma_{i+1}$ executes a behavioral node n_i defined by AGL. Since the composed concern DSLs do not change the AGL behavioral state and only restrict when a node is enabled (via guards) and which states are admissible (via invariants), each UDML step corresponds to an admissible AGL step on the projected states. Hence, projecting the UDML execution by π_{AGL} yields a valid AGL execution.

Completeness. Conversely, consider an AGL execution that satisfies all enablement restrictions and invariants induced by the composed concerns. Starting from an initial UDML snapshot σ_0 with $\pi_{AGL}(\sigma_0) = q_0$, each AGL step can be realized as a UDML transition because the same node is enabled and the resulting snapshot remains admissible.

Repeating this argument step by step constructs a UDML execution whose projection matches the given AGL execution.

Therefore, along executions, UDML snapshots and AGL snapshots are in stepwise one-to-one correspondence via π_{AGL}.

5. Transforming UDML to Event-B

This section focuses on transforming the UDML model to the Event-B environment for the purposes of simulation and verification of unified domain models. We define a semantics-preserving compilation scheme that translates incorporated UDML models into Event-B contexts and machines.

The UDML2Evnet-B transformation basically is defined by the following mapping follows. Structural elements and global constraints are translated into Event-B *contexts*, while executable behaviour and security enforcement are translated into Event-B *machines*. This separation aligns with the Event-B modeling discipline and supports modular reasoning about domain structure, behaviour, and access control.

Mapping Structural Elements. Structural components defined using DCSL—including domain classes, attributes, associations, and structural invariants—are mapped to Event-B contexts. Domain classes are represented as carrier sets, attributes and associations as variables or constants with appropriate typing invariants, and structural constraints are translated into Event-B invariants. This mapping establishes the static domain state space over which behaviour is executed.

Mapping Behavioral Models. Behaviour specified using AGL is mapped to Event-B machines. Each activity graph is translated into a machine whose state variables encode the current control state of the graph. Activity graph nodes are mapped to Event-B events, and edges determine the control-flow relations between events. The firing of an event corresponds to the execution of the associated activity node, resulting in a state transition that reflects both control-flow progression and updates to the domain state induced by the corresponding module actions.

Mapping RBACDom Constraints. Security concerns specified using RBACDom are integrated directly into the behavioral semantics through event guards and invariants. Each `RbacDomNode` that corresponds to an AGL node (via `aglNode`) is translated into guard predicates on the corresponding Event-B event. These guards encode the node-enablement conditions derived from role requirements, permission checks, and SoD constraints. In this way, authorization is enforced as a prerequisite for event execution rather than as an external post hoc check.

SoD constraints that span multiple nodes or sessions are translated into Event-B invariants, ensuring that all reachable states of the machine satisfy the specified security properties. This guarantees that RBACDom constraints are preserved across all executions and refinements of the model.

Table 3 summarises the mapping rules between UDML constructs and their corresponding Event-B artefacts.

From this perspective, the executable semantics of a UDML model is defined by the behavior of the generated Event-B contexts and machines as well as the proof obligations they induce.

Algorithm 1 realises the semantics-preserving translation defined in Section 4 and treats UDML as a unifying integration layer for structural, behavioral, and security concerns.

Proposition 1 (Deny-Overrides Semantics). *Let $n \in ND$ be a behavioral node and let s be a session state. If $combiningAlg(R) =$* denyOverrides *and there exists an applicable RBACDom annotation $rna \in App(n, s)$*

Table 3. Mapping from UDML to Event-B

UDML Construct	Event-B Artefact	Description
DomainModel	Context	Defines the static universe of discourse, including carrier sets and global constants.
Concern	Context (structural grouping)	Used to organise constants, axioms, and invariants related to a specific modeling concern.
Domain Class (DCSL)	Carrier Set / Constant	Each domain class is mapped to a carrier set or abstract set representing its instances.
Attribute (DCSL)	Variable / Function	Attributes are mapped to state variables or functions whose domains and ranges are constrained by invariants.
Association (DCSL)	Relation / Invariant	Associations are mapped to relations between carrier sets, with multiplicity constraints enforced as invariants.
Activity Graph (AGL)	Machine	Each activity graph is mapped to an Event-B machine capturing its execution semantics.
Initial Node	Initialisation Event	The initial node of an activity graph determines the initial value of the control-state variable.
Node (AGL)	Event	Each node is mapped to an Event-B event representing an atomic execution step.
Edge (AGL)	Guard	Edges are mapped to guards that constrain which node-events may fire based on the current control state.
Control State (current node)	Variable	The currently active node is represented by a state variable in the machine.
ModuleAct	Event Action	Module actions are translated into Event-B substitutions that update the machine state.
RbacDomNode	Event Guard	RBAC constraints specified by `RbacDomNode` (corresponding to AGL nodes) are mapped to guards enforcing role and permission requirements.
Required Roles	Guard Predicate	Role requirements are encoded as predicates over the active roles of the current session.
Required Permissions	Guard Predicate	Permission requirements are encoded as predicates derived from role-to-permission assignments.
Session	Variable	The current execution context is represented as a session variable.
Separation-of-Duty Constraint	Invariant	SoD constraints are encoded as invariants restricting role assignment or role activation.
Policy (ALL_OF / ANY_OF)	Guard Logic	Policy semantics determine whether conjunction or disjunction is used in RBAC guard predicates.
Effect (ALLOW / DENY)	Guard Semantics	The effect determines whether the guard enables or blocks event execution.
Rule Combining Strategy	Guard Construction	Authorization guards are synthesized according to the strategy selected by $combiningAlg(R)$, including denyOverrides, permitOverrides, and firstApplicable.
Well-Formedness Rules	Invariants / Axioms	Well-formedness constraints are translated into invariants or axioms to ensure model consistency.

such that $effect(rna) = DENY$, *then* $Enable_{RBAC}(n, s) = false$.

Proof. By the definition of $Dec(n, s)$ under the denyOverrides strategy, the existence of an applicable annotation $rna \in App(n, s)$ with $effect(rna) = DENY$ implies $Dec(n, s) = DENY$. Hence, $Enable_{RBAC}(n, s) \triangleq Dec(n, s) = ALLOW$ does not hold. Therefore, $Enable_{RBAC}(n, s) = false$.

The translation is organised into successive phases, each corresponding to a clearly identified block of lines in the algorithm.

Model initialization. Lines 1–4 check model well-formedness, initialise the Event-B context and machine, and link the machine to the context to access structural definitions.

Structural translation. Lines 5–9 compile the DCSL structural specification: domain

classes become carrier sets, while attributes and associations are translated into constants and invariants, defining the static domain universe.

Behavioral translation. Lines 10–15 translate AGL behavior. Carrier sets and constants represent activity graphs, nodes, and control flow. A control-state variable *pc* tracks the active node. Nodes and edges are compiled into events with guards encoding control flow and actions implementing module behavior.

Security translation. Lines 16–22 compile the RBACDom specification. Users, roles, permissions, and sessions are mapped to sets and constants; assignments and role activation to variables; and RBAC constraints, including SoD, to invariants.

Cross-concern incorporation. Lines 23–26 integrate behavior and security via a partial resource mapping for late binding. Derived predicates for enabled roles and permissions are introduced to simplify guard construction.

Rule applicability. Lines 27–45 define rule applicability and construct authorization guards based on $combiningAlg(M.\mathcal{R})$. Under denyOverrides, execution requires at least one applicable allow and no applicable deny rule. Under permitOverrides, any applicable allow suffices. Under firstApplicable, the decision follows the first applicable annotation according to *ord*.

Initialization and verification. Lines 46–50 generate the *INITIALISATION* event to establish a consistent state, then produce and discharge proof obligations using Rodin. Successful discharge ensures that the generated Event-B model preserves UDML semantics and enforces the specified security policies.

Verification of RBACDom-enriched UDML models is performed using standard Event-B proof obligations generated by Rodin. Table 4 summarizes the main PO categories and their roles in ensuring safety, executability, refinement correctness, and liveness.

In the current case study, the rule-combining strategy is instantiated using the denyOverrides policy. This choice reflects a conservative security principle commonly adopted in RBAC systems, where denying policies take precedence over allowing ones.

However, the proposed semantic framework does not fundamentally depend on this particular strategy. As defined in Subsection 4.1.3 and Subsection 4.1.4, rule combination is parameterized through the function $combiningAlg : DomainModelRbacDom \rightarrow CA$. Consequently, alternative strategies such as permitOverrides and firstApplicable can be specified directly at the modeling level without modifying the UDML-to-Event-B transformation pipeline.

6. Tool Support and Evaluation

This section reports the tool-supported workflow used to operationalize and evaluate the proposed semantic framework for UDML. Rodin is used as a verification back-end to discharge POs of the generated Event-B models, thereby validating the formal semantics of UDML models composed with RBACDom.

6.1. Experiment Configuration

This subsection describes the experimental setup used in the evaluation. We evaluate feasibility and effectiveness on a case study based on the Open Journal Systems (OJS) domain. Figure 7 depicts the OJS submission management process, modeled as an AGL activity graph in UDML and composed with RBACDom policies via node-level correspondences.

We consider a fixed set of seven roles:
Roles = {Author, Reviewer, JournalManager, Editor, Copyeditor, LayoutEditor, Proofreader}.
In total, nine node-level RBACDom policies $\langle P_{N1}, \ldots, P_{N9} \rangle$ are specified as `RbacDomNode` instances inside the RBACDom model and associated with the OJS submission management process steps by correspondence to AGL node

Algorithm 1 UDML-to-Event-B translation and verification

Input: A well-formed UDML model $M = \langle \mathcal{U}, \mathcal{S}, \mathcal{B}, \mathcal{R}, C \rangle$.
Output: An Event-B development $\mathcal{D}(M) = \langle C, \mathsf{Mach} \rangle$ and proof statistics.

Global conventions:
$\mathcal{T}_D$ — translation functions for DCSL (structure)
$\mathcal{T}_A$ — translation functions for AGL (behaviour)
$\mathcal{T}_S$ — translation functions for RBACDom (security)
$\mathsf{WF}(\cdot)$ — well-formedness checker (UDML + DCSL + AGL + RBACDom)
$Ann(n)$ — the set of `RbacDomNode` annotations associated with node n

assert $\mathsf{WF}(M)$
Initialise an empty Event-B *context* $C \leftarrow \langle \mathsf{SETS}, \mathsf{CONSTS}, \mathsf{AXMS} \rangle$
Initialise an empty Event-B *machine* $\mathsf{Mach} \leftarrow \langle \mathsf{VARS}, \mathsf{INVS}, \mathsf{EVTS} \rangle$
Link Mach to C (i.e., Mach **sees** C)
$(\mathsf{SETS}_D, \mathsf{CONSTS}_D, \mathsf{AXMS}_D) \leftarrow \mathcal{T}_D^{ctx}(M.\mathcal{S})$
$C.\mathsf{SETS} \leftarrow C.\mathsf{SETS} \cup \mathsf{SETS}_D$
$C.\mathsf{CONSTS} \leftarrow C.\mathsf{CONSTS} \cup \mathsf{CONSTS}_D$
$C.\mathsf{AXMS} \leftarrow C.\mathsf{AXMS} \cup \mathsf{AXMS}_D$
$\mathsf{Mach.INVS} \leftarrow \mathsf{Mach.INVS} \cup \mathcal{T}_D^{inv}(M.\mathcal{S})$
$C.\mathsf{SETS} \leftarrow C.\mathsf{SETS} \cup \{\mathtt{NODES}, \mathtt{EDGES}, \mathtt{GRAPHS}\}$
$C.\mathsf{CONSTS} \leftarrow C.\mathsf{CONSTS} \cup \{src, tgt, nodesOf, edgesOf, initOf\}$
$C.\mathsf{AXMS} \leftarrow C.\mathsf{AXMS} \cup \mathcal{T}_A^{ctx}(M.\mathcal{B})$
Add machine variables for execution: $\mathsf{Mach.VARS} \leftarrow \mathsf{Mach.VARS} \cup \{curG, pc\}$
Add behavioral invariants: $\mathsf{Mach.INVS} \leftarrow \mathsf{Mach.INVS} \cup \mathcal{T}_A^{inv}(M.\mathcal{B}, curG, pc)$
Generate behavioral events: $\mathsf{Mach.EVTS} \leftarrow \mathsf{Mach.EVTS} \cup \mathcal{T}_A^{evt}(M.\mathcal{B}, curG, pc)$
$(\mathsf{SETS}_S, \mathsf{CONSTS}_S, \mathsf{AXMS}_S) \leftarrow \mathcal{T}_S^{ctx}(M.\mathcal{R})$
$C.\mathsf{SETS} \leftarrow C.\mathsf{SETS} \cup \mathsf{SETS}_S$ $C.\mathsf{CONSTS} \leftarrow C.\mathsf{CONSTS} \cup \mathsf{CONSTS}_S$ $C.\mathsf{AXMS} \leftarrow C.\mathsf{AXMS} \cup \mathsf{AXMS}_S$
Add constants $permAction \in \mathtt{PERMISSIONS} \to \mathtt{ACTIONS}$ and $permRes \in \mathtt{PERMISSIONS} \to \mathtt{RESOURCES}$
Add axioms typing $permAction$, $permRes$
Add RBAC state variables: $\mathsf{Mach.VARS} \leftarrow \mathsf{Mach.VARS} \cup \mathcal{T}_S^{vars}(M.\mathcal{R})$
Add RBAC invariants (including SoD): $\mathsf{Mach.INVS} \leftarrow \mathsf{Mach.INVS} \cup \mathcal{T}_S^{inv}(M.\mathcal{R})$
Add late-binding for protected resources: add constant (or variable) $mapRes \in \mathtt{RESOURCES} \rightharpoonup \mathtt{AE}$
Add axiom/invariant $\mathsf{Typing}(mapRes)$
Define derived predicates used in guards: $EnabledRoles(sess) \triangleq active(sess)$
$EnabledPerms(sess) \triangleq \{p \mid \exists r \in active(sess) \cdot (r, p) \in pa\}$
Let $alg \leftarrow combiningAlg(M.\mathcal{R})$
for all node-execution event $ev_n \in \mathsf{Mach.EVTS}$ corresponding to node $n \in ND$ **do**
  Let $Allow \leftarrow \emptyset$ and $Deny \leftarrow \emptyset$
  for all $rna \in Ann(n)$ **do**
    Compute $Applies(rna, sess) \equiv RolesSat(rna, sess) \wedge PermsSat(rna, sess) \wedge SoDSat(rna, sess)$
    if $effect(rna) = ALLOW$ **then**
      $Allow \leftarrow Allow \cup \{Applies(rna, sess)\}$
    else
      $Deny \leftarrow Deny \cup \{Applies(rna, sess)\}$
    end if
  end for
  Define $AllowApplies(n, sess) \equiv \bigvee Allow$
  Define $DenyApplies(n, sess) \equiv \bigvee Deny$
  if $alg =$ denyOverrides **then**
    Strengthen guard(ev_n) with $AllowApplies(n, sess) \wedge \neg DenyApplies(n, sess)$
  else if $alg =$ permitOverrides **then**
    Strengthen guard(ev_n) with $AllowApplies(n, sess) \vee (\neg AllowApplies(n, sess) \wedge \neg DenyApplies(n, sess) \wedge Ann(n) = \emptyset)$
  else
    Let $first(n, sess)$ be the applicable annotation in $App(n, sess)$ with minimal order ord
    Strengthen guard(ev_n) with $(App(n, sess) \neq \emptyset \wedge effect(first(n, sess)) = ALLOW)$
  end if
end for
Generate *INITIALISATION* to initialise all machine variables consistently with invariants
set $curG$ to the selected graph instance (or default graph) set $pc \leftarrow initOf(curG)$
initialise ua, pa, $sess$, $active$ (and any domain state variables) so that all invariants hold
Generate proof obligations $PO(\mathcal{D}(M))$ (invariant preservation, feasibility, well-definedness, etc.)
Discharge $PO(\mathcal{D}(M))$ using Rodin provers and record proof statistics
return $(\mathcal{D}(M), PO\text{-statistics})$

Table 4. PO Categories for RBACDom-Enriched UDML Models

PO Category	Event-B PO Type	Purpose	Verified Property
Well-definedness	WD	Ensures that all expressions in guards, actions, and invariants are well-typed and mathematically defined	Semantic soundness
Safety: invariant preservation, event feasibility	INV, FIS	Guarantees invariant preservation and feasibility of state transitions under structural, behavioral, and RBAC constraints	Safety, consistency
Refinement correctness: guard strengthening, simulation obligations, witness-related obligations	GRD, SIM, WIT	Ensures that refined events correctly simulate abstract behavior and preserve refinement relations	Behavioral correctness
Progress (when applicable): variant-decrease and well-foundedness obligations	VAR, NAT	Proves progress and termination of control steps using well-founded variants for convergent events	Liveness (progress)
Deadlock freedom: theorem obligations	THM	Establishes that at least one event is enabled in all admissible domain states	Executability

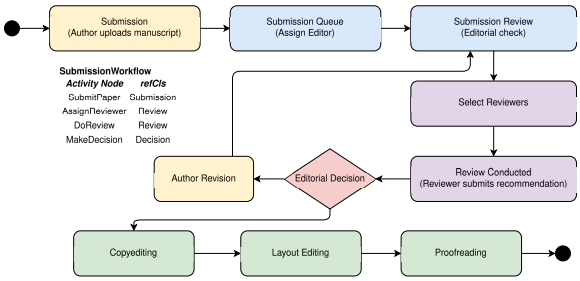

Figure 7. The OJS submission management process.

labels (via `aglNode`). Additional details are given in Appendix 8.1.

6.2. Mapping RBAC Policies to AGL

Each policy P_{Ni} is specified as a node-level access-control rule in RBACDom and represented by a `RbacDomNode`. A policy constrains the execution of a particular submission management process step by inducing an authorization condition over the corresponding AGL node identified by `aglNode`. Concretely, a policy specifies: (i) authorized roles, (ii) object-scoped predicates over domain entities, (iii) the policy mode and effect, and (iv) optional SoD and compliance constraints. The resulting policies for the OSJ submission management process are summarized in Table 5.

This node-level specification localizes access control at behavioral execution boundaries while preserving separation between DCSL, AGL and RBACDom.

The formal specification is built on top of a static context comprising four fundamental carrier sets: *USERS*, representing system actors; *SUBMISSIONS*, denoting data objects; *NODES*, capturing the process topology; and *ROLES*, corresponding to functional responsibilities.

6.3. Experimental Results

To prevent invalid states, the system defines a set of invariants that serve as a mathematical safety net, ranging from basic typing constraints to DSD policies. As a representative example, the business rule *"An author must not assume the role of reviewer for their own submission"* is formalized as the following mandatory predicate:

$$\begin{aligned} &\forall u, s \cdot (u \mapsto (s \mapsto r\_Author)) \in assignment \\ &\Rightarrow (u \mapsto (s \mapsto r\_Reviewer)) \notin assignment \end{aligned} \quad (1)$$

The correctness of the resulting Event-B model is validated using the ProB model checker within the Rodin platform, with the verification results illustrated in Figure 8.

As shown in Figure 8, the verification process is visualized through three interface areas. The

Table 5. Node-Correspondence RBACDom Policies for OJS

Activity Node	**Policy**	**RBACDom constraint (roles, scope, SoD, compliance)**
SubmitPaper	`<<rbac:P_N1>>`	{roles={Author}, policy=ALL_OF, effect=ALLOW, scope=owns(u, *sub*) by construction (author field set to u), SoD=SSD/DSD(Author≠Reviewer on *sub*, Author≠Editor on *sub*), compliance=log(createSubmission), hideReviewerIdentities}
AssignEditor	`<<rbac:P_N2>>`	{roles={JournalManager}, policy=ALL_OF, effect=ALLOW, scope=state(*sub*) = SUBMITTED, SoD=DSD(JournalManager≠Author on *sub*) [optional], compliance=log(assignEditor)}
EditorialCheck	`<<rbac:P_N3>>`	{roles={Editor}, policy=ALL_OF, effect=ALLOW, scope=assignedEditor(u, *sub*) ∧ state(*sub*) = SUBMITTED, SoD=DSD(Editor≠Author on *sub*), compliance=log(editorialCheck)}
SelectReviewer	`<<rbac:P_N4>>`	{roles={Editor}, policy=ALL_OF, effect=ALLOW, scope=assignedEditor(u, *sub*) ∧ state(*sub*) ∈ {EDITORIAL_CHECKED, UNDER_REVIEW}, SoD=DSD(Editor≠Author on *sub*)∧DSD(Editor≠Reviewer on *sub*), compliance=log(assignReviewer), enforceBlindReview}
DoReview	`<<rbac:P_N5>>`	{roles={Reviewer}, policy=ALL_OF, effect=ALLOW, scope=assignedReviewer(u, *sub*) ∧ state(*sub*) = UNDER_REVIEW ∧ ¬coi(u, *sub*), SoD=DSD(Reviewer≠Author on *sub*)∧DSD(Reviewer≠Editor on *sub*), compliance=log(submitReview), deadlineLogged}
EditorialDecision (Decision node)	`<<rbac:P_N6>>`	{roles={Editor}, policy=ALL_OF, effect=ALLOW, scope=assignedEditor(u, *sub*) ∧ state(*sub*) = UNDER_REVIEW ∧ allReviewsSubmitted(*sub*), SoD=DSD(Editor≠Author on *sub*)∧DSD(Editor≠Reviewer on *sub*), compliance=log(editorDecision), rationaleRequired [optional]}
CopyEditing	`<<rbac:P_N7>>`	{roles={Copyeditor}, policy=ALL_OF, effect=ALLOW, scope=assignedCopyeditor(u, *sub*) ∧ state(*sub*) = ACCEPTED, SoD=DSD(Copyeditor≠Author on *sub*) [optional], compliance=log(copyedit), noAccessToReviewData}
LayoutEditing	`<<rbac:P_N8>>`	{roles={LayoutEditor}, policy=ALL_OF, effect=ALLOW, scope=assignedLayoutEditor(u, *sub*) ∧ state(*sub*) = COPYEDITED, SoD=DSD(LayoutEditor≠Copyeditor on *sub*) [optional], compliance=log(layout), checksumVersioning [optional]}
Proofreading	`<<rbac:P_N9>>`	{roles={Proofreader}, policy=ALL_OF, effect=ALLOW, scope=assignedProofreader(u, *sub*) ∧ state(*sub*) = LAYOUT_DONE, SoD=DSD(Proofreader≠LayoutEditor on *sub*) [optional], compliance=log(proofApproval), readOnlyContent}

left panel presents the structural relationship between the Event-B context and machine, while the right panel records a concrete execution trace of a paper from initialization to completion, demonstrating that the process is deadlock-free. The central panel (State View) provides explicit evidence of safety: the green "T" (True) indicators confirm that, throughout the execution scenario, all security constraints—including the DSD rule in Equation (1)—are preserved.

The generated Event-B model, comprising context and machine components, is imported into Rodin, where proof obligations are automatically generated and discharged to validate executability, security constraints, and cross-concern consistency.

The composed UDML model is then systematically translated into Event-B to enable tool-supported verification.

The verification statistics reported in Table 6 are derived from the Event-B artefacts generated for the OJS submission management process composed with RBACDom. These artefacts include the structural domain model, the behavioral activity graphs, and the RBACDom security constraints introduced in the unified UDML model. The reported numbers are obtained by classifying the invariants and

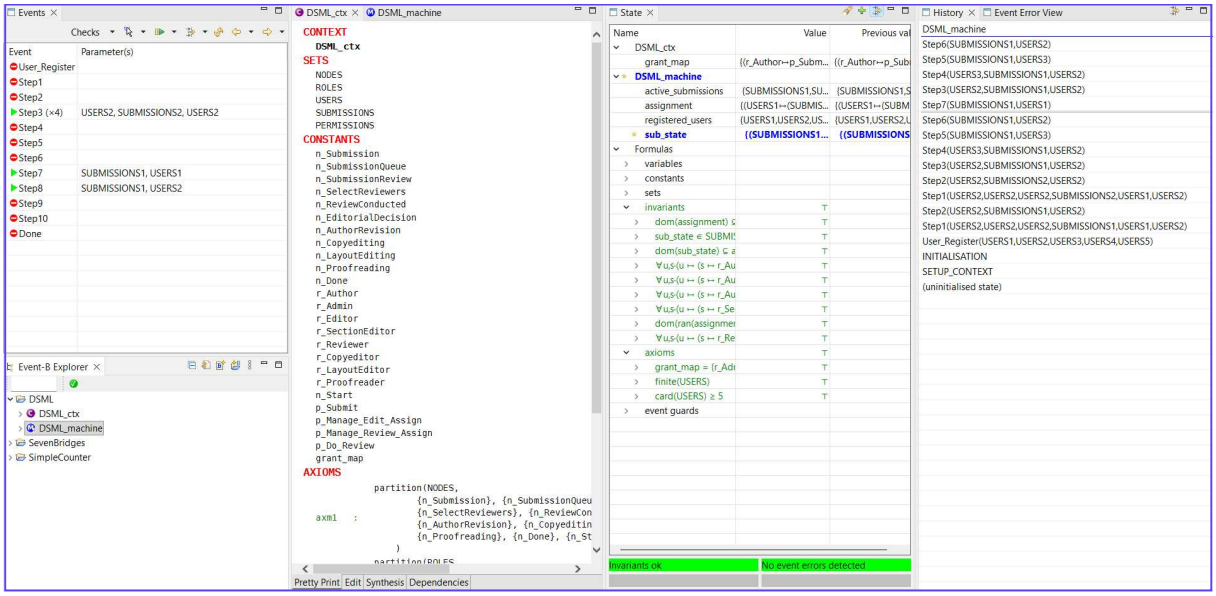

Figure 8. Experimental verification in Rodin/ProB using the OJS system as a case study.

events generated during the UDML-to-Event-B translation and applying the standard Event-B proof-obligation generation rules.

The derivation distinguishes five groups of verification conditions, corresponding to different modelling concerns: (i) core RBAC semantics, (ii) role assignment and separation-of-duty constraints, (iii) runtime authorization semantics for sessions and role activation, (iv) cross-concern bindings between RBAC constraints and OJS behavioral steps via RBACDom annotations, and (v) context- and state-based constraints governing the submission lifecycle. For each group, the number of invariants and events determines the resulting proof obligations according to Event-B verification semantics. Table 6 summarizes this derivation.

To assess the verification effort, we analyze the distribution of proof obligations by discharge status. Table 7 summarizes the results produced by the Rodin platform.

Proof automation. In total, 381 proof obligations were generated, of which 369 (96.8%) were automatically discharged by the built-in Rodin provers, while the remaining 12 (3.2%) required minor interactive proof steps.

Most automatically discharged obligations correspond to well-definedness (WD) and guard strengthening (GRD) conditions (Table 4), which are efficiently handled by the automated provers. The remaining interactive proofs mainly concern invariant preservation and required only simple reasoning steps, such as selecting relevant hypotheses or instantiating quantified variables. No complex proof tactics or structural modifications were necessary; all remaining obligations were discharged using the built-in predicate prover.

Note that not all Event-B proof obligation types listed in Table 4 arise in this case study. In particular, refinement-related obligations (FIS, SIM, WIT) and variant-based obligations (VAR, NAT) are not generated, as the current model does not include refinement steps or convergent events. Therefore, Table 7 reports only the proof obligation types effectively produced by the Rodin platform.

On liveness properties. The current evaluation

Table 6. Derivation of verification statistics from OJS and RBACDom formal artifacts

Model Component	**Formal Basis**	**# Invariants**	**# Events**	**# Proof Obligations**
Core RBAC Model	Role–permission consistency, mutual exclusion, and RBAC well-formedness constraints derived from RBACDom, independent of the OJS workflow.	14	8	96
Assignment & SoD Constraints	Static and dynamic separation-of-duty constraints derived from RBACDom specifications for Authors, Reviewers, Editors, and production roles.	11	6	78
Runtime Authorization (Sessions)	Session and role-activation semantics ensuring that only assigned and active roles may execute protected operations.	9	5	64
Cross-Concern Binding (RBAC × OJS)	Binding invariants linking RBAC authorization constraints with OJS workflow steps and domain states through RBACDom annotations.	12	7	102
Context / State-Based Constraints	Workflow-ordering and state-evolution constraints restricting admissible domain transitions in the OJS lifecycle.	6	4	41
Overall	—	**52**	**30**	**381**

Table 7. Distribution of proof obligations and discharge status

PO Type	**Total**	**Auto**	**Manual**
WD	120	120	0
INV	140	132	8
GRD	80	78	2
THM	41	39	2
Total	**381**	**369**	**12**

primarily focuses on safety properties, including invariant preservation, authorization correctness, and deadlock freedom. These properties ensure that the system does not reach undesirable states during execution. Event-B also supports reasoning about liveness properties through variant functions and convergence proofs. In particular, progress towards a target state can be established by defining a variant that decreases with each execution step.

To illustrate this, consider a simplified workflow with four states: `Start`, `Review`, `Decision`, and `Done`. A liveness property can be expressed informally as follows: every submission eventually reaches the `Done` state. Since Event-B does not directly support temporal operators such a ”eventually”, this property can be established indirectly by defining a ranking function $rank : State \rightarrow \mathbb{N}$, for example: $rank(Start) = 0$, $rank(Review) = 1$, $rank(Decision) = 2$, $rank(Done) = 3$. A corresponding variant can then be defined as: $V(s) = 3 - rank(s)$. Under this encoding, each transition that advances the workflow strictly decreases the value of V, thereby establishing progress towards the terminal state `Done`. If all relevant events are proved convergent with respect to this variant, termination of the workflow follows.

However, in the presence of cyclic control-flow structures or authorization constraints that may re-enable earlier states, the variant may fail to decrease monotonically, making liveness proofs more challenging. Such situations may indicate potential design issues, including unintended infinite loops or blocked execution paths.

For this reason, the present evaluation does not claim a full proof of liveness for the OJS workflow. Rather, it focuses on safety and executability, while a systematic treatment of liveness properties, including automated variant

synthesis and convergence analysis, is left for future work.

6.4. Discussion

Rodin is used in this paper to validate the proposed semantic framework rather than to benchmark tool performance. Automated proof-obligation (PO) generation and discharge enable the early detection of semantic inconsistencies and policy violations that would otherwise remain implicit in executable domain models.

The OJS results confirm that UDML models composed with RBACDom can be interpreted as unified transition systems with formally analyzable executability and security properties. In this setting, AGL defines the behavioral evolution of the domain model by specifying when a node can be executed according to the control-flow semantics, while RBACDom determines whether the node is authorized to execute. The reported proof statistics further indicate that the verification effort is largely handled by standard Event-B mechanisms, achieving a high degree of proof automation without requiring ad hoc verification extensions.

Another important design aspect is the rule-combining strategy used to evaluate multiple RBACDom annotations attached to the same behavioral node. In the revised framework, rule combination is treated as a configurable model-level parameter through the function $combiningAlg : DomainModelRbacDom \rightarrow CA$. This allows different authorization semantics, including denyOverrides, permitOverrides, and firstApplicable, to be specified directly within the RBACDom model while preserving the same UDML-to-Event-B transformation scheme. This extension improves the generality of the framework without affecting the structural semantics of DCSL or the behavioral semantics of AGL.

Limitations and usability. The current evaluation focuses on a realistic but moderate-sized process with a fixed role set; broader generalization requires additional case studies. Moreover, modelling choices such as the granularity of node decomposition and the expressiveness of scope predicates directly influence the size of the generated Event-B model and the number of proof obligations.

Furthermore, the high proportion of automatically discharged proof obligations in the OJS case study (96.8%) indicates that most generated obligations correspond to standard Event-B proof obligations that are efficiently handled by the Rodin provers, providing initial evidence that the approach can scale without a proportional increase in manual verification effort. Nevertheless, a comprehensive evaluation on large-scale industrial systems remains future work.

The current prototype uses annotation-based specifications to associate RBACDom policies with behavioral nodes. While this representation enables a straightforward integration with the existing code base, manually writing annotations may become error-prone for large workflows such as the OJS process. Future work will therefore focus on providing graphical and model-driven tool support for RBACDom, enabling security policies to be specified directly at the domain-model level and automatically translated into the corresponding annotations and formal artefacts. Such tool support would reduce manual specification effort and improve usability for larger-scale systems.

From a language-engineering perspective, the annotation mechanism used in the current prototype can also be interpreted as an annotation-based domain-specific language (aDSL). In this view, RBACDom annotations embedded in the domain model constitute an internal DSL integrated into the host object-oriented programming language (e.g., Java). Following our approach presented in [8], each concern-specific DSL defined at the modelling level corresponds to an equivalent annotation-based representation in the host language. This

mapping establishes a one-to-one correspondence between the external DSL used in the domain model and its internal DSL representation embedded in the program. Consequently, the unified domain model can be interpreted as an executable specification whose semantics can be realized through model transformation and code-generation mechanisms. From this perspective, annotations such as `@RBACDomNode` do not merely serve as syntactic metadata. Instead, they represent elements of the aDSL embedded in the host programming environment. This design allows domain constraints and security policies to remain closely aligned with the domain model while preserving executability within the host-language ecosystem.

7. Related Works

This work is situated at the intersection of (i) Domain-Driven Design (DDD) and role-based access control (RBAC), (ii) concern-oriented domain-specific languages (DSLs) for structural and behavioral modeling, and (iii) refinement-based formal verification using Event-B.

7.1. Incorporating Concerns into Domain Models

Domain-Driven Design (DDD) advocates domain models as central artefacts for capturing domain knowledge and ensuring alignment between domain experts and software developers. To support this goal, a wide range of modeling approaches—most notably UML, OCL, and domain-specific languages (DSLs)—have been employed to express domain structure, behaviour, and constraints [5, 6]. Within this paradigm, correctness is not limited to data integrity but also encompasses behavioral consistency and compliance with domain rules.

In systems with non-trivial authorization requirements, access control becomes an intrinsic part of domain behaviour rather than a peripheral technical concern [20]. Several studies have emphasized that treating security policies as external or implementation-level artefacts leads to mismatches between domain behaviour and authorization logic, thereby complicating maintenance and undermining correctness [19, 21, 22]. From a DDD perspective, RBAC policies must therefore be reflected explicitly at the domain-modeling level [23, 24].

Formal techniques such as Alloy [18, 25], Petri nets [26], and structural operational semantics [27] provide mathematically rigorous foundations for verification. However, these approaches are rarely integrated into DDD- or DSL-based modeling workflows and do not directly support the concern-oriented, annotation-based composition mechanism required by UDML [8]. Similarly, existing research in DSL engineering has largely focused on the design and tooling of individual DSLs [6], without offering a unified semantic framework capable of guaranteeing correctness simultaneously across structural, behavioral, and security concerns.

However, existing DDD-oriented approaches typically integrate RBAC in an informal or ad hoc manner. The work in [23] reports that RBAC integration in multi-domain systems often lacks formal grounding, relying instead on descriptive models or implementation-level mechanisms. Similarly, [22] proposes DSL-based modeling of RBAC for complex systems but focuses primarily on expressiveness rather than formal semantics or verification. As a result, while these approaches improve modularity and readability, they do not provide guarantees of semantic consistency or correctness across domain concerns.

7.2. DSLs to Represent Domain Models

To enhance modeling capabilities, particularly for expressing RBAC concerns within the domain model, several DSLs have been proposed [5, 6]. DSLs facilitate the explicit representation of business constraints, policies, and roles, and provide a solid foundation for code

generation, testing, and system maintenance. Among existing approaches, DCSL [14], AGL [17], and UDML [8] are particularly relevant, as they focus on modeling both the structural and behavioral aspects of the domain, while also addressing cross-cutting concerns such as security. DCSL is a highly expressive DSL that supports the automatic generation of behavioral specifications and software artifacts through an annotation-based mechanism. Similarly, AGL is an internal DSL designed to integrate behavioral aspects—such as UML activity diagrams—into a unified domain model using a concise and expressive syntax. DCSL and AGL concentrate on domain modeling and do not explicitly address the formal verification of SSD/DSD constraints.

In our recent work, UDML [8] further advances domain modeling by unifying multiple domain concerns within a single modeling language, enabling the development of complex systems with a high degree of expressiveness. Nevertheless, UDML currently lacks a formal semantic foundation for verifying whether integrated RBAC concerns satisfy security constraints. Although UDML models can be translated into OCL constraints or annotation-based code, verification is limited to static analysis and does not provide formal proofs of conflict freedom.

In contrast, our proposed approach is novel in that it augments UDML with a formal semantics and a proof-based verification mechanism. By grounding RBAC as a concern-specific DSL within UDML and providing a formal verification backend, our method ensures that RBAC policies are safe, conflict-free, and preserved throughout the software lifecycle.

7.3. Event-B based Verification of Unified Domain Models

Formal methods have long been used to verify safety and security properties of complex systems, with Event-B providing a particularly well-established framework based on refinement, invariants, and proof obligations [25, 28]. Several studies have applied Event-B to access control verification by encoding RBAC concepts as invariants and authorization rules as guards [29, 30]. Other work has explored the combination of DSLs with Event-B to enable design-time verification of security policies [12, 18].

While these approaches demonstrate the effectiveness of Event-B for verifying access control, they typically focus on standalone policy models or security-specific DSLs. The integration with broader domain models—particularly those aligned with DDD principles—remains limited. Moreover, authorization is often treated as a separate verification problem rather than as an intrinsic constraint on executable domain behaviour.

In contrast, our approach uses Event-B as a semantic back-end for UDML. A semantics-preserving mapping translates unified domain models—comprising structure, behaviour, and RBAC constraints—into Event-B contexts and machines. Authorization constraints directly restrict behavioral transitions via guards and invariants, allowing proof obligations to capture both domain correctness and security properties within a single formal framework.

This work addresses these gaps by extending UDML with a formally defined RBAC concern, integrating authorization constraints directly into executable domain behaviour, and providing a semantics-preserving mapping to Event-B for proof-based verification. As a result, the proposed approach offers a unified and verifiable foundation for concern-oriented executable domain models, going beyond existing integration and verification techniques.

8. Conclusions

This paper addresses the construction of unified domain models that remain semantically well-defined and formally verifiable in the presence of multiple concerns. Within the

context of Domain-Driven Design, we propose a semantic framework for UDML that clarifies the operational semantics of domain models incorporating structural, behavioral, and security aspects.

As a concrete security concern, we introduced RBACDom as an annotation-based DSL for representing RBAC within unified domain models. We further defined a transformation from UDML models enriched with RBACDom into Event-B, enabling proof-based verification of executability, cross-concern consistency, and correctness properties using the Rodin platform.

The feasibility of the approach was demonstrated through a realistic case study based on the Open Journal Systems (OJS) submission management process. The successful verification of the generated Event-B models, with a high degree of proof automation, indicates that the approach is applicable to non-trivial domain models and suitable for model-driven development workflows.

Future work will extend the framework toward additional concern-specific DSLs and more expressive executable domain models, including dynamic and context-dependent authorization. Further enhancements to tool support and evaluations on larger-scale and industrial case studies are planned to assess scalability, usability, and practical adoption. In addition, while the current evaluation focuses on safety and executability, the framework is compatible with Event-B techniques for reasoning about liveness via variants and convergence, which will be explored in future work.

References

[1] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.

[2] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, 2nd Edition, Morgan&Claypool, 2017.

[3] Q. Ramadan, M. Salnitriy, D. Struber, J. Jurjens, P. Giorgini, From Secure Business Process Modeling to Design-Level Security Verification, in: 2017 ACM/IEEE 20th Int. Conf. Model Driven Engineering Languages and Systems (MODELS), IEEE, Austin, TX, 2017, pp. 123–133.

[4] M. Roggenbach, A. Cerone, B.-H. Schlingloff, G. Schneider, S. A. Shaikh, Correction to: Formal Methods for Software Engineering, in: M. Roggenbach, A. Cerone, B.-H. Schlingloff, G. Schneider, S. A. Shaikh (Eds.), Formal Methods for Software Engineering: Languages, Methods, Application Domains, Springer International Publishing, Cham, 2022, pp. C1–C1.

[5] M. Fowler, Domain-Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.

[6] Andrzej Wasowski, Thorsten Berger, Domain-Specific Languages Effective Modeling, Automation, and Reuse, Springer Cham, 2023.

[7] A. Wasowski, T. Berger, Internal Domain-Specific Languages, in: Domain-Specific Languages: Effective Modeling, Automation, and Reuse, Springer, Cham, 2023, pp. 357–394.

[8] V.-V. Le, D.-H. Dang, An Approach to Composing Concerns for an Executable Unified Domain Model, in: 2024 rivf Int. Conf. Computing and Communication Technologies (RIVF), 2024, pp. 424–428.

[9] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, Cambridge, 2010.

[10] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, International Journal on Software Tools for Technology Transfer, Vol. 12, No. 6, 2010, pp. 447–466, https://doi.org/10.1007/s10009-010-0145-y.

[11] D. Mery, The Event B Modelling Method, Erasmus (2011).

[12] A. Rawat, R. S. Suryavanshi, V. Nagar, D. Yadav, Formal Development of Blockchain Enabled E Healthcare System Using Event-B, Available at SSRN 5167558 (2025).

[13] Public Knowledge Project, Open journal systems (ojs), https://pkp.sfu.ca/software/ojs/ (2024).

[14] D. M. Le, D.-H. Dang, V.-H. Nguyen, On domain driven design using annotation-based domain specific language, Computer Languages, Systems & Structures, Vol. 54, 2018, pp. 199–235.

[15] V. C. Hu, R. Kuhn, D. Yaga, Verification and test methods for access control policiesmodels, Tech. Rep. NIST SP 800-192, National Institute of Standards and Technology, Gaithersburg, MD (Jun. 2017).

[16] ANSI INCITS 359-2012 Information Technology – Role Based Access Control (2012).

[17] D.-H. Dang, D. M. Le, V.-V. Le, AGL: Incorporating behavioral aspects into domain-driven design, Information and Software Technology, Vol. 163, 2023, pp. 107284.

[18] I. Vistbakka, E. Troubitsyna, Towards Integrated Modelling of Dynamic Access Control with UML and Event-B, Electronic Proceedings in Theoretical Computer Science, Vol. 271, 2018, pp. 105–116, https://doi.org/10.4204/EPTCS.271.8.

[19] OMG, Unified Modeling Language 2.5.1, Object Management Group, 2017.

[20] K. Sohr, M. Drouineaud, G.-J. Ahn, M. Gogolla, Analyzing and Managing Role-Based Access Control Policies, IEEE Transactions on Knowledge and Data Engineering, Vol. 20, No. 7, 2008, pp. 924–939.

[21] D. Basin, J. Doser, T. Lodderstedt, Model driven security: From UML models to access control infrastructures, ACM Trans. Softw. Eng. Methodol., Vol. 15, No. 1, 2006, pp. 39–91, https://doi.org/10.1145/1125808.1125810.

[22] O. Oruc, Role-Based Embedded domain-specific language for collaborative multi-agent systems through blockchain technology, in: 9th Int. Conf. Security, Privacy and Trust Management (SPTM), 2021, pp. 1–19.

[23] Y. Li, Z. Du, Y. Fu, L. Liu, Role-based access control model for inter-system cross-domain in multi-domain environment, Applied Sciences, Vol. 12, No. 24, 2022, pp. 13036.

[24] F. Gonzalez-Lopez, G. Bustos, J. Munoz-Gama, M. Sepulveda, Domain model based design of business process architectures, Applied Sciences, Vol. 12, No. 5, 2022, pp. 2563.

[25] M. Kherbouche, B. Molnar, Formal Model Checking and Transformations of Models Represented in UML with Alloy, in: Modelling to Program, Springer International Publishing, Cham, 2021, pp. 127–136.

[26] B. Shafiq, A. Masood, J. Joshi, A. Ghafoor, A Role-Based Access Control Policy Verification Framework for Real-Time Systems, in: 10th IEEE Inte. Work. Object-Oriented Real-Time Dependable Systems, IEEE, Sedona, AZ, USA, 2005, pp. 13–20.

[27] F. P. M. Stappers, M. A. Reniers, S. Weber, Transforming SOS Specifications to Linear Processes, in: Formal Methods for Industrial Critical Systems, Vol. 6959, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 196–211.

[28] W. Mallouli, J.-M. Orset, A. Cavalli, N. Cuppens, F. Cuppens, A formal approach for testing security rules, in: Proc. 12th ACM symposium on Access control models and technologies - SACMAT '07, ACM Press, Sophia Antipolis, France, 2007, p. 127, https://doi.org/10.1145/1266840.1266860.

[29] F. Akeel, A. Salehi Fathabadi, F. Paci, A. Gravell, G. Wills, Formal Modelling of Data Integration Systems Security Policies, Data Science and Engineering, Vol. 1, No. 3, 2016, pp. 139–148, https://doi.org/10.1007/s41019-016-0016-y.

[30] I. Mendil, Y. Ait-Ameur, N. K. Singh, G. Dupont, D. Mery, P. Palanque, Formal domain-driven system development in Event-B: Application to interactive critical systems, Journal of Systems Architecture, Vol. 135, 2023, pp. 102798.

APPENDIX

Structural well-formedness rules of the RBACDom metamodel

This section defines the *structural well-formedness rules (WFRs)* of the RBACDom metamodel. These rules are specified using the OCL and are attached directly to the RBACDom metamodel elements. Their purpose is to ensure that RBAC specifications are structurally valid, well-typed, and unambiguous before any semantic interpretation or formal verification is performed.

WF-S1: Uniqueness of RBAC identifiers

```
context User
inv WF_S1_UserIdIsUnique:
  User.allInstances()->isUnique(u |u.id)

context Role
inv WF_S1_RoleIdIsUnique:
  Role.allInstances()->isUnique(r|r.id)

context Permission
inv WF_S1_PermissionIdIsUnique:
  Permission.allInstances()->isUnique(p|p.id)

context Action
inv WF_S1_ActionIdIsUnique:
  Action.allInstances()->isUnique(a|a.id)

context ProtectedResource
inv WF_S1_ResourceIdIsUnique:
  ProtectedResource.allInstances()
  ->isUnique(r | r.id)
```

WF-S2: Well-typed role assignments

```
context RoleAssignment
inv WF_S2_RoleAssignmentWellTyped:
  self.user <> null and self.role <> null
```

WF-S3: Well-typed permission assignments

```
context PermissionAssignment
inv WF_S3_PermissionAssignmentWellTyped:
  self.role <> null and self.permission <> null
```

WF-S4: Valid permission definition

```
context Permission
inv WF_S4_PermissionIsActionResourcePair:
  self.action <> null and self.resource <> null
```

WF-S5: Unique permission tuples

```
context Permission
inv WF_S5_UniquePermissionTuples:
  Permission.allInstances()
->isUnique(p | Tuple{action = p.action,
resource = p.resource})
```

WF-S6: Resource-domain element type compatibility

```
context ProtectedResource
inv WF_S6_OptionalResourceBinding:
  self.mapsTo = null or
  (
(self.type = ResourceType::CLASS and
self.mapsTo.kind = kind::CLASS)
or (self.type = DomainType::METHOD and
self.mapsTo.kind = DomainType::METHOD)
or(self.type = ResourceType::ATTRIBUTE and
self.mapsTo.kind = DomainType::ATTRIBUTE)
or(self.type = ResourceType::ENTITY and
self.mapsTo.kind = DomainType::ENTITY)
or(self.type = ResourceType::API and
(self.mapsTo.kind = DomainType::METHOD
or
self.mapsTo.kind = DomainType::ENTITY))
  )
```

WF-S7: Resource–domain type compatibility

```
context ProtectedResource
inv WF_S7_ResourceDomainTypeCompatibility:
  self.mapsTo <> null implies
  (
    (self.type = ResourceType::CLASS and
Set{DomainType::CLASS}->includes(self.mapsTo.kind))
or(self.type = ResourceType::METHOD and
Set{DomainType::METHOD}->includes(self.mapsTo.kind))
or(self.type = ResourceType::ATTRIBUTE and
Set{DomainType::ATTRIBUTE}->includes(self.mapsTo.kind))
or(self.type = ResourceType::ENTITY and
Set{DomainType::ENTITY}->includes(self.mapsTo.kind))
or(self.type = ResourceType::API and
Set{DomainType::METHOD, DomainType::ENTITY}
        ->includes(self.mapsTo.kind))
  )
```

WF-S8: Well-formed separation of duty constraints

```
context SoDConstraint
inv WF_S8_MinimumRoleSetAndCardinality:
  self.conflictingRoles->size()
  >= 2 and self.cardinality >= 2
```

WF-S9: Static separation of duty consistency

```
context SSDConstraint
inv WF_S9_SSDNotViolatedStatically:
  User.allInstances()->forAll(u |
    u.assignments->collect(a | a.role)
      ->intersection(self.conflictingRoles)
      ->size() < self.cardinality
  )
```

WF-S10: Annotation attachment and scope

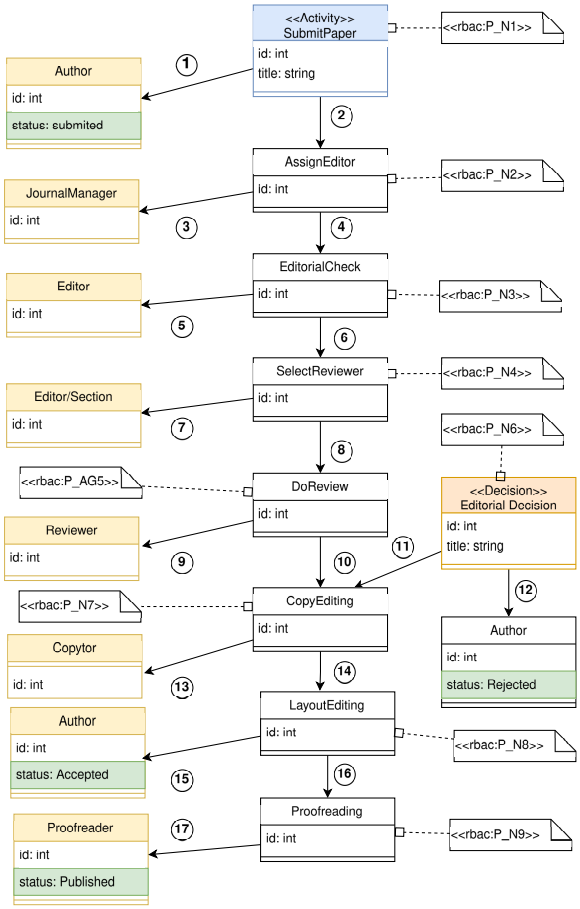

Figure 9. UDML activity graph of the OJS submission management process with node-correspondent RBAC policies.

```
context AnnotationRbac
inv WF_S10_AnnotationMustHaveTarget:
  self.target <> null
```

WF-S11: Deterministic ordering for firstApplicable

```
context DomainModelRbacDom
inv WF_S11_FirstApplicableOrdering:
 self.combiningAlg=CombiningAlgKind::firstApplicable
 implies self.rbacDomNodes->forAll(a1, a2|a1<>a2 and
 a1.targetNode=a2.targetNode implies a1.ord<>a2.ord)
```

```
@DClass
public class Student {
    @DAttr(id = true, auto = true)
    private int id;
    ...
}
```

8.1. Activity Graph of the OJS Submission Management Process

Figure 9 illustrates the OJS submission workflow as a UDML activity graph $g = \langle N, E, src, tgt, init \rangle$. Each node $n \in N$ represents a domain execution step and is linked to the structural model via `refCls`, providing the context for execution and state evolution.

The workflow starts at *SubmitPaper*, where a `Submission` is created, and proceeds through *AssignEditor* and *EditorialCheck*, followed by reviewer selection and evaluation (*SelectReviewer*, *DoReview*). A decision node (*EditorialDecision*) directs execution toward acceptance—with production steps such as *CopyEditing* and *LayoutEditing*—or rejection and termination.

Access control is integrated by composing RBACDom with the activity graph via node-level correspondences. Selected nodes are annotated with `RbacDomNode`, specifying authorization constraints (roles, scope, SoD). These annotations preserve the control flow while restricting node enabledness.

Thus, the annotated activity graph defines a unified model where AGL governs control flow and RBACDom prunes unauthorized transitions, forming the behavioral and security basis for the unified executable semantics.

8.2. Specifying RBAC for Each User Role

This appendix specifies the RBACDom configuration used in the OJS case study. For each user role, we define the corresponding role identifier, the controlled actions and protected resources, the permitted action–resource pairs, together with scope constraints, separation-of-duty (SoD) constraints, prohibitions, and audit requirements.

The specifications below define the security domain model (`DomainModelRbacDom`) that is composed with the behavioral model through

node-level correspondences, as described in the main text. They are independent of behavioral control flow and are referenced by node-anchored `RbacDomNode` specifications.

U1 – Author Role identifier: `Author`.
Actions. {CreateSubmission, SubmitSubmission, UploadManuscriptFile, ReplaceManuscriptFile, CreateMetadata, UpdateMetadata, DeleteMetadata, UploadRevision, RespondToReviews, ViewOwnSubmission, ViewDecisionLetter, ViewReviewSummary, UploadSupplementaryFile, DeleteSupplementaryFile}.
Protected resources. {Submission, ManuscriptFile, RevisionFile, SubmissionMetadata, SupplementaryFile, ReviewReport_Anonymised, DecisionLetter, AuthorResponse, ReviewerIdentity}.
Permissions. Permitted ⟨action, resource⟩ pairs include submission creation and update, metadata management, revision handling, and read-only access to decision letters and anonymised review summaries.
Scope constraints. Actions on an existing submission *sub* require $owns(u, sub)$. Metadata updates and manuscript replacement are restricted to DRAFT or REVISION_REQ states. Revision-related actions are allowed only in REVISION_REQ.
Separation of duty. An author of a submission cannot act as its reviewer or editor (SSD/DSD).
Prohibitions. Access to reviewer identities is denied. Editorial and review actions are prohibited.
Audit. Submission creation, updates, and revisions are logged with user, object, timestamp, and version information.

U2 – Reviewer Role identifier: `Reviewer`.
Actions. {ViewAssignedSubmission, ViewAnonymisedMetadata, CreateReview, UpdateReview, SubmitReview, UploadReviewAttachment, ViewReviewDeadline}.
Protected resources. {Submission, SubmissionMetadata_Anonymised, Review, ReviewAttachment, ReviewReport, DecisionLetter, ReviewerIdentity, AuthorIdentity}.
Permissions. Review creation, update, and submission on assigned submissions.
Scope constraints. Reviewer actions require $assignedReviewer(u, sub)$ and $state(sub)$ = UNDER_REVIEW. Only anonymised metadata is visible.
Separation of duty. A reviewer cannot be the author or editor of the same submission (DSD).
Prohibitions. Access to author identities and editorial actions is denied.
Audit. Review activities are logged with deadlines and submission timestamps.

U3 – Editor Role identifier: `Editor`.
Actions. Editorial checking, reviewer assignment, review inspection, decision creation and finalisation, and author notification.
Protected resources. {Submission, Review, ReviewReport, ReviewAssignment, Decision, DecisionLetter, ReviewerIdentity, AuthorIdentity}.
Scope constraints. Editorial actions require $assignedEditor(u, sub)$. Decisions require all reviews to be submitted.
Separation of duty. Editors cannot act as authors or reviewers for the same submission (DSD).
Prohibitions. Editors cannot perform production editing or submit reviews.
Audit. Reviewer assignments and editorial decisions are logged with rationale.

U4 – Journal Manager Role identifier: `JournalManager`.
Actions and resources. Journal configuration, user and role management, and audit-log access.
Scope constraints. Actions apply at journal level only.
Separation of duty. Journal managers cannot act as editors or reviewers (SSD/DSD).
Audit. All administrative changes are logged.

U5 – Copyeditor Role identifier: `Copyeditor`.
Scope constraints. Copyediting is allowed only for accepted submissions.
Separation of duty. A copyeditor cannot be the author of the same submission.
Prohibitions. Access to review data and editorial actions is denied.

U6 – Layout Editor Role identifier: `LayoutEditor`.
Scope constraints. Layout actions are allowed only after copyediting.
Separation of duty. Layout editors cannot act as copyeditors on the same submission.

U7 – Proofreader Role identifier: `Proofreader`.
Scope constraints. Proofreading is allowed only after layout completion.
Separation of duty. Proofreaders cannot act as layout editors for the same submission.